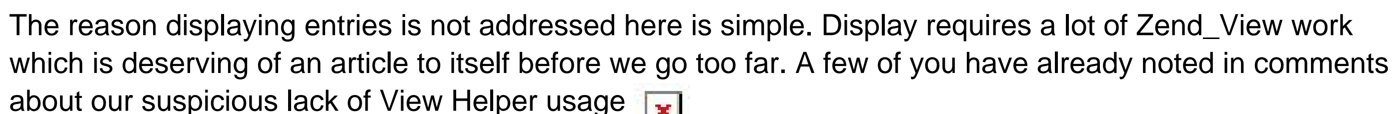
Wednesday, May 14. 2008

Example Zend Framework Blog Application Tutorial: Part 8: Creating and Editing Blog Entries with a dash of HTMLPurifier

There's nothing quite like having a functioning application emerge out of the controlled chaos we know as The Development Process. In Part 8 of the ongoing saga describing how to build a real world blog application using the Zend Framework we finally reach the point at which we concentrate on blog entries. At the end of this Part, we will be able to create and edit entries in preparation for Part 9 when we will explore displaying them to the world!

Previously: Example Zend Framework Blog Application Tutorial - Part 7: Authorisation with Zend_Acl and Revised Styling

The reason displaying entries is not addressed here is simple. Display requires a lot of Zend_View work which is deserving of an article to itself before we go too far. A few of you have already noted in comments about our suspicious lack of View Helper usage



. By now a few uncomfortable flaws should be becoming apparent in how some template content is becoming duplicated.

So, on with the show already!

Step 1: Adding an Entry Controller and Add Action Template

The first step to creating new entries will be writing an Admin_EntryController class (the prefix is needed since it's situated in the Admin Module) with an addAction() method and matching template.

The template will utilise a new Zend_Form object for gathering the input used to create a new entry, as well as offering an editing View.

We'll start with the Controller, added at `/application/admin/controllers/EntryController.php`:

```
<?php
class Admin_EntryController extends Zend_Controller_Action
{
   public function addAction()
   {}
   public function listAction()
   {}
   public function editAction()
   {}
   public function deleteAction()
   {}
}
```

The EntryController needs four basic methods. We intend creating, editing and deleting entries, as well as listing all entries to an Author to select those options.

We'll concentrate on the addAction() method first, so let's add an appropriate template at `/application/admin/views/scripts/entry/add.phtml` which has an old friend referred to:

```
<p>Where I am not understood, it shall be concluded that something very useful and profound is couched underneath.<br /><em>- Jonathan Swift</em></p>
<?php if($this->failedValidation): ?>
  <p class="error">Some problems were detected with the submitted form.</p>
<?php endif; ?>
<?php echo $this->entryForm ?>
```

Let's accompany our blog entry forms with a little Jonathan Swift for inspiration . We once again meet a validation problem message identical to the one in our login form. As we're duplicating this message we're going to have to consider a means later on of isolating such commonly used feedback messages for reuse somewhere in Part 9.

To keep our styling synchronised with our intended form output, add the following to `/public/css/style.css`:

```
textarea {
  width: 76%;
  height: 30em;
}
```

This should provide a decent styling for textareas for most browsers. Perhaps even Safari which I noticed recently is displaying some of my text fields poorly.

Step 2: Assembling the Entry Form with Zend_Form

We've already had a fairly detailed look at Zend_Form back in Part 6 when we wrote a subclass to contain some standard and specific decorator arrays for form elements, and created our Login Form example. The same principles used there also apply here with very few changes. Once again we are using the shorter array-based syntax over multiple method calls. One of the differences to take note of is that I've used two new options `attribs` and `value` which no form developer could live without!

We'll start with a new class called ZFBlog_Form_EntryAdd located at `/library/ZFBlog/Form/EntryAdd.php`:

```
<?php
class ZFBlog_Form_EntryAdd extends ZFBlog_Form
{
  public function init()
  {
    $this->setAction('/admin/entry/add');
    // Display Group #1 : Entry Data

    $this->addElement('text', 'title', array(
      'decorators' => $this->_standardElementDecorator,
      'label' => 'Title:',
      'attribs' => array(
        'maxlength' => 200,
        'size' => 80
```

```
    ),
    'validators' => array(
        array('StringLength', false, array(3,200))
    ),
    'required' => true
));

$this->addElement('text', 'date', array(
    'decorators' => $this->_standardElementDecorator,
    'label' => 'Date:',
    'attribs' => array(
        'maxlength' => 16,
        'size' => 16
    ),
    'value' => Zend_Date::now()->toString('yyyy-MM-dd HH:mm'),
    'validators' => array(
        array('Date', false, array('yyyy-MM-dd HH:mm', 'en'))
    ),
    'required' => true
));

$this->addElement('textarea', 'entrybody', array(
    'decorators' => $this->_standardElementDecorator,
    'label' => 'Entry Body:',
    'required' => true
));

$this->addElement('textarea', 'entrybodyextended', array(
    'decorators' => $this->_standardElementDecorator,
    'label' => 'Extended Body:'
));

$this->addDisplayGroup(
    array('title','date','entrybody','entrybodyextended'), 'entrydata',
    array(
        'disableLoadDefaultDecorators' => true,
        'decorators' => $this->_standardGroupDecorator,
        'legend' => 'Entry'
    )
);

// Display Group #2 : Submit
$this->addElement('submit', 'submit', array(
    'decorators' => $this->_buttonElementDecorator,
    'label' => 'Save'
));
$this->addDisplayGroup(
    array('submit'), 'entrydatasubmit',
    array(
        'disableLoadDefaultDecorators' => true,
        'decorators' => $this->_buttonGroupDecorator,
        'class' => 'submit'
    )
```

```
        );
    }
}
```

I threw in a sprinkling of Zend_Date above to format the current date to a format compatible with a MySQL database. Zend_Date is also used in the background by the Zend_Validator_Date class which is why the date formatting is identical to both. Note that the formatting is not the PHP regular style used by the date() function, but instead uses ISO format specifiers. This offers a great deal of flexibility if you want something other than MySQL formatted dates.

Finally note that there is no "required" flag for the "entrybodyextended" element since an extended body is optional.

We're not quite done yet. Our two textareas, as discussed way back in Part 1, are being designed to accept HTML input since I really can't be bothered to play with custom formatting tags and such. This obviously raises the risk that I, another author, or someone who's guessed my password may, either by mistake or intent, add some Cross-Site Scripting (XSS) into the mix and introduce a devastating security exploit. We can't have that now!

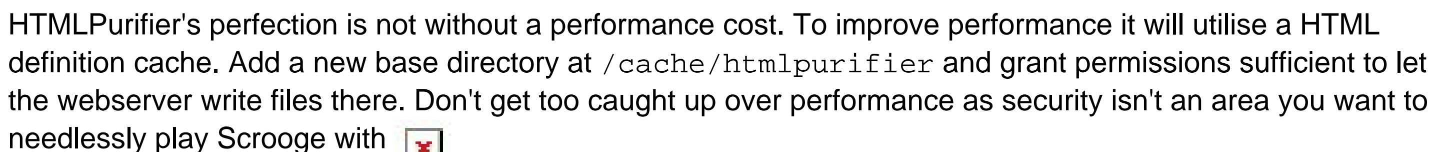
Step 3: Filtering Entries using a HTMLPurifier based Custom Filter

What we will do now is attach a custom filter to the textareas containing our Entry data that cleans up any HTML input, removes XSS, and as a bonus converts any non-HTML input into formatted HTML. This, for example, covers our text paragraphs by wrapping them in <p> tags.

My favourite library for achieving this is HTMLPurifier which I consider one of those much under utilised libraries in PHP. To my knowledge, there is nothing out there to beat its comprehensive feature list. Its finest feature is that it actually understands HTML across multiple standards. Input is tokenised, parsed, passed through a whitelist (the opposite of a detection blacklist), reformed as perfectly correct valid output, and then it can optionally wrap stuff like plain text in paragraphs. All for your preferred DTD. Either you're using HTMLPurifier, or you are using something second or third rate, and I have no qualms whatsoever in stating that as a fact. I cannot praise this library more highly.

To start, you'll need to download a copy of HTMLPurifer 3.1.0rc1 (which is the latest release at the time of writing) and copy the contents of the package's `/library` directory into our blog application's `/library`. Since HTMLPurifer follows the PEAR Convention for class naming and file location, we need make no changes to our include_path. If you prefer, you can also install HTMLPurifer from its PEAR channel as described on the download page. Although it's a release candidate I haven't run into any problems using it other than my unexplainable habit of mispelling HTMLPurifier as HTMLPurifer which has led to a few frustrating hair pulling moments!

HTMLPurifier's perfection is not without a performance cost. To improve performance it will utilise a HTML definition cache. Add a new base directory at `/cache/htmlpurifier` and grant permissions sufficient to let the webserver write files there. Don't get too caught up over performance as security isn't an area you want to needlessly play Scrooge with



. The performance cost is really only noticeable if using it for post-processing of output being sent to visitors. The cache coupled with the fact HTMLPurifier is used only in the backend Administration eliminates any such cost for the purposes of our application.

Let's introduce our custom filter classes. I'm saving them using a mirror directory structure of the Zend

Framework as usual. Here a standard one I usually use with HTMLPurifier saved to `/library/ZFBlog/Filter/HTMLPurifier.php`:

```
<?php
class ZFBlog_Filter_HTMLPurifier implements Zend_Filter_Interface
{

   protected $_htmlPurifier = null;

   public function __construct($options = null)
   {
      $config = null;
      if (!is_null($options)) {
         $config = HTMLPurifier_Config::createDefault();
            foreach ($options as $option) {
               $config->set($option[], $option[1], $option[2]);
            }
      }
      $this->_htmlPurifier = new HTMLPurifier($config);
   }
   public function filter($value)
   {
      return $this->_htmlPurifier->purify($value);
   }
}
```

HTMLPurifier options have three distinct elements we can pass to this filter as an array. We could stop here, passing filter options for every form, but this is a very general filter class whose sole purpose is to pass options into HTMLPurifier, so let's add a subclass of this specifically for HTML textual input with predefined options at `/library/ZFBlog/Filter/HtmlBody.php`:

```
<?php
class ZFBlog_Filter_HtmlBody extends ZFBlog_Filter_HTMLPurifier
{

   public function __construct($newOptions = null)
   {
      $options = array(
         array('Cache', 'SerializerPath',
            Bootstrap::$root . '/cache/htmlpurifier'
         ),
         array('HTML', 'Doctype', 'XHTML 1.0 Strict'),
         array('HTML', 'Allowed',
            'p,em,h1,h2,h3,h4,h5,strong,a[href],ul,ol,li,code,pre,'
            .'blockquote,img[src|alt|height|width],sub,sup'
         ),
         array('AutoFormat', 'Linkify', 'true'),
         array('AutoFormat', 'AutoParagraph', 'true')
      );

      if (!is_null($newOptions)) {
         // I'll let HTMLPurifier overwrite original options
```

```
            // with new ones rather than filter them myself
            $options = array_merge($options, $newOptions);
        }

        parent::__construct($options);
    }
}
```

This new subclass passes specific options to HTMLPurifier. We provide the path to the cache directory created previously, inform the library our output should conform to XHTML 1.0 Strict, add a whitelist of allowed tags and attributes, and finally enable two optional formatting helpers to auto paragraph output (wrapped with `<p>` tags) and transform URLs into hyperlinks. If ZFBlog_Filter_HtmlBody needs further adjustment we can pass it options when attaching this filter to our form elements.

This really is how HTMLPurifer works. By using sensible defaults, configuration before use is extremely simple.

With our two custom filters in tow, we now need to make sure the Zend Framework can actually find them! We've done this previously actually when registering a custom decorator path with Zend_Form. Let's repeat the process. Here's an updated ZFBlog_Form class from `/library/ZFBlog/Form.php`:

```
<?php
class ZFBlog_Form extends Zend_Form
{
    protected $_standardElementDecorator = array(
        'ViewHelper',
        array('LabelError', array('escape'=>false)),
        array('HtmlTag', array('tag'=>'li'))
    );
    protected $_buttonElementDecorator = array(
        'ViewHelper'
    );
    protected $_standardGroupDecorator = array(
        'FormElements',
        array('HtmlTag', array('tag'=>'ol')),
        'Fieldset'
    );
    protected $_buttonGroupDecorator = array(
        'FormElements',
        'Fieldset'
    );

    protected $_noElementDecorator = array(
        'ViewHelper'
    );
    public function __construct($options = null)
    {
        // Path setting for custom classes MUST ALWAYS be first!
        $this->addElementPrefixPath('ZFBlog_Form_Decorator', 'ZFBlog/Form/Decorator/', 'decorator');
        $this->addElementPrefixPath('ZFBlog_Filter', 'ZFBlog/Filter/', 'filter');
        $this->_setupTranslation();
        parent::__construct($options);
```

```
        $this->setAttrib('accept-charset', 'UTF-8');
        $this->setDecorators(array(
            'FormElements',
            'Form'
        ));
    }
    protected function _setupTranslation()
    {
        if (self::getDefaultTranslator()) {
            return;
        }
        $path = Bootstrap::$root . '/translate/forms.php';
        $translate = new Zend_Translate('array', $path, 'en');
        self::setDefaultTranslator($translate);
    }
}
```

Now Zend_Form can use our custom filters. The last thing we do is attach our new HtmlBody custom filter to our new form along with a few other filters for good measure:

```
<?php
class ZFBlog_Form_EntryAdd extends ZFBlog_Form
{

    public function init()
    {
        $this->setAction('/admin/entry/add');
        // Display Group #1 : Entry Data

        $this->addElement('text', 'title', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Title:',
            'attribs' => array(
                'maxlength' => 200,
                'size' => 80
            ),
            'validators' => array(
                array('StringLength', false, array(3,200))
            ),
            'filters' => array('StringTrim'),
            'required' => true
        ));

        $this->addElement('text', 'date', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Date:',
            'attribs' => array(
                'maxlength' => 16,
                'size' => 16
            ),
            'value' => Zend_Date::now()->toString('yyyy-MM-dd HH:mm'),
            'validators' => array(
```

```
            array('Date', false, array('yyyy-MM-dd HH:mm', 'en'))
        ),
        'required' => true
    ));

    $this->addElement('textarea', 'entrybody', array(
        'decorators' => $this->_standardElementDecorator,
        'label' => 'Entry Body:',
        'filters' => array('HtmlBody'),
        'required' => true
    ));

    $this->addElement('textarea', 'entrybodyextended', array(
        'decorators' => $this->_standardElementDecorator,
        'label' => 'Extended Body:',
        'filters' => array('HtmlBody')
    ));

    $this->addDisplayGroup(
        array('title','date','entrybody','entrybodyextended'), 'entrydata',
        array(
            'disableLoadDefaultDecorators' => true,
            'decorators' => $this->_standardGroupDecorator,
            'legend' => 'New Entry'
        )
    );

    // Display Group #2 : Submit
    $this->addElement('submit', 'submit', array(
        'decorators' => $this->_buttonElementDecorator,
        'label' => 'Save'
    ));
    $this->addDisplayGroup(
        array('submit'), 'entrydatasubmit',
        array(
            'disableLoadDefaultDecorators' => true,
            'decorators' => $this->_buttonGroupDecorator,
            'class' => 'submit'
        )
    );
  }
}
```

Let's get this form attached to a View now.

Step 4: Output Form from Add Action Template

Let's revise our EntryController to pass our sparkling new form to the addAction() method's associated template.

```
<?php
```

```
class Admin_EntryController extends Zend_Controller_Action
{
    public function addAction()
    {
        $form = new ZFBlog_Form_EntryAdd;
        if (!$this->getRequest()->isPost()) {
            $this->view->entryForm = $form;
            return;
        } elseif (!$form->isValid($_POST)) {
            $this->view->failedValidation = true;
            $this->view->entryForm = $form;
            return;
        }
    }
}
```

Now, keeping in mind we should keep track of what error messages need to be cleaned up into a shorter message (as I noted previously, I like short error messages attached to element labels), let's add the Zend_Validate_Date default errors for replacement in our translation array in `/translate/forms.php`:

```
<?php
return array(
    Zend_Validate_NotEmpty::IS_EMPTY => 'Required',
    Zend_Validate_StringLength::TOO_SHORT => 'Minimum Length of %min%',
    Zend_Validate_StringLength::TOO_LONG => 'Maximum Length of %max%',
    Zend_Validate_Date::NOT_YYYY_MM_DD => 'Must use YYYY-MM-DD format',
    Zend_Validate_Date::INVALID => 'Not valid date',
    Zend_Validate_Date::FALSEFORMAT => 'Invalid date format',
);
```

Let's give it a whirl! Open up `http://zfblog/admin/entry/add` in your browser (you may need to login as Joe Bloggs first with the username "joebloggs" and password "password" setup earlier) and marvel at the new form.

We're still not done. If a valid form is submitted we're going to need to store it to the database!

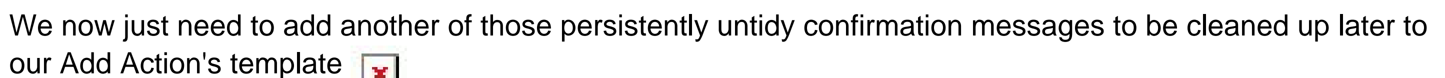
Step 5: Storing New Enties to the Database

We've already put in place an Entries Model when we were setting up our database, so saving entries is just a simple addition to our EntryController.

```
<?php
class Admin_EntryController extends Zend_Controller_Action
{
    public function addAction()
    {
        $form = new ZFBlog_Form_EntryAdd;
        if (!$this->getRequest()->isPost()) {
            $this->view->entryForm = $form;
            return;
```

```
        } elseif (!$form->isValid($_POST)) {
            $this->view->failedValidation = true;
            $this->view->entryForm = $form;
            return;
        }

        $values = $form->getValues();
        $table = new Entries;
        $data = array(
            'title' => $values['title'],
            'date' => $values['date'],
            'author' => Zend_Auth::getInstance()->getIdentity()->username,
            'author_id' => Zend_Auth::getInstance()->getIdentity()->id,
            'body' => $values['entrybody'],
            'extended_body' => $values['entrybodyextended']
        );
        $table->insert($data);
        $this->view->entrySaved = true;
    }
}
```

I've left the class as is, but at some point I'll refactor this. One of the few worthwhile aesthetic goals of any source code is to keep methods as short as possible. Almost half the above method is mapping form names to database field names. If used a more direct convention, we could have saved reading space and had a shorter method.

We now just need to add another of those persistently untidy confirmation messages to be cleaned up later to our Add Action's template 
.

```
<p>Where I am not understood, it shall be concluded that something very useful and profound is couched underneath.<br /><em>- Jonathan Swift</em></p>
<?php if($this->failedValidation): ?>
    <p class="error">Some problems were detected with the submitted form.</p>
<?php endif; ?>
<?php if($this->entrySaved): ?>
    <p class="success">New entry has been saved.</p>
<?php else: ?>
    <?php echo $this->entryForm ?>
<?php endif; ?>
```

Go ahead and try out the new entry addition. As noted in the introduction we'll cover the actual display of Entries in the next Part, so for now check out the results on the database using PhpMyAdmin or similar. You notice paragraphs are correctly wrapped with <p> and links linkified as HTMLPurifier makes it's presence felt. Tags and attributes that were not added to our whitelist will also be stripped.

Step 6: Editing Entries

Editing entries is a fairly simple addition to our zoo. In fact, it's necessary. We already have in place a form

class for inputting entries and it's only a small step from there to add in the old values for editing.

The only hoop to jump through is reversing the two default formatting steps that HTMLPurifier performs before entries are saved to the database. This can be a confusing problem because many people, I suspect, assume populating Zend_Form with values will output those values exactly. Not so. If you have added Filters to a form object you have actually told Zend_Form that only values passing this filter are entirely valid, so guess what? Yes, populating data is filtered when added to a form object. Since here we filter all entry body data through HTMLPurifier, while the actual input from the user is not, we need to disable this filter before displaying the populated edit form.

There is another strategy you could also use. Instead of applying the filter to the form, apply it instead within the Model using overridden save() and update() methods. I've elected not to do this since I quite like having the filtering as part of the form object itself. It doesn't change the fact that that hoop exists, but locating it in the Model could aid in reuse since it's now decoupled from a form object and managed closer to the database. But anyway, that's what Refactoring is for! So let's forge ahead.

We'll start with implementing the listAction() for our Entry Controller so we can view a list of all entries for editing/deletion. For the moment this does not include paging.

```php
<?php
class Admin_EntryController extends Zend_Controller_Action
{
   public function addAction()
   {
      $form = new ZFBlog_Form_EntryAdd;
      if (!$this->getRequest()->isPost()) {
         $this->view->entryForm = $form;
         return;
      } elseif (!$form->isValid($_POST)) {
         $this->view->failedValidation = true;
         $this->view->entryForm = $form;
         return;
      }

      $values = $form->getValues();
      $table = new Entries;
      $data = array(
         'title' => $values['title'],
         'date' => $values['date'],
         'author' => Zend_Auth::getInstance()->getIdentity()->username,
         'author_id' => Zend_Auth::getInstance()->getIdentity()->id,
         'body' => $values['entrybody'],
         'extended_body' => $values['entrybodyextended']
      );
      $table->insert($data);
      $this->view->entrySaved = true;
   }

   public function listAction()
   {
      $table = new Entries;
      // SELECT title, date, id FROM entries
      $rows = $table->fetchAll(
```

```
        $table->select()->from($table, array('title','date','id'))
    );
    $this->view->entries = $rows->toArray();
  }

  public function editAction()
  {}

  public function deleteAction()
  {}
}
```

The associated view simply iterates the array of entries over a foreach construct to replicate an entry list.

```
/application/admin/views/scripts/entry/list.phtml
<?php if (count($this->entries) > ): ?>
  <ul class="entry-list">
  <?php foreach($this->entries as $entry): ?>
    <li>
      <a href="/admin/entry/edit/id/<?php echo $entry['id'] ?>">
        <?php echo $entry['title'] ?></a>
        - <a href="/admin/entry/edit/id/<?php echo $entry['id'] ?>">Edit</a>
        | <a href="/admin/entry/delete/id/<?php echo $entry['id'] ?>">Delete</a>
    </li>
  <?php endforeach; ?>
  </ul>
<?php else: ?>
  <p class="error">There are no entries for this blog.</p>
<?php endif; ?>
```

It looks pretty darn ugly, but will fit for now. Logged in as an Author, you can view the list of existing entries assuming you've added a few test ones while testing our new add entry functionality. Each will have a link to edit or delete that entry. We'll cover the URL form I used shortly.

Let's create a new form for editing entries. Luckily we have the Entry Add form, so all we need do is subclass this to make a few small adjustments! Add this form as `/library/ZFBlog/Form/EntryEdit.php`:

```
<?php
class ZFBlog_Form_EntryEdit extends ZFBlog_Form_EntryAdd
{

  public function init()
  {
    parent::init();
    $this->setAction('/admin/entry/edit');

    // What entry id are we editing?!
    $this->addElement('hidden', 'id', array(
      'decorators' => $this->_noElementDecorator,
      'validators' => array(
        'Digits'
```

```
        ),
        'required' => true
    ));

    $this->getDisplayGroup('entrydata')->setLegend('Edit Entry');
    $this->getDisplayGroup('entrydata')->addElement(
        $this->getElement('id')
    );
  }

}
```

Isn't a little reuse worthwhile?

Before displaying our Edit Form we need to populate it with the data we're editing. This involves capturing the entry from the database, reversing any HTMLPurifier filtering, disabling the Form's filter chain, and only then re-populating the form. We need to be careful the workflow only disables filters for form display - they should remain in force for any submitted edits.

```
<?php
class Admin_EntryController extends Zend_Controller_Action
{
  public function addAction()
  {
    $form = new ZFBlog_Form_EntryAdd;
    if (!$this->getRequest()->isPost()) {
        $this->view->entryForm = $form;
        return;
    } elseif (!$form->isValid($_POST)) {
        $this->view->failedValidation = true;
        $this->view->entryForm = $form;
        return;
    }

    $values = $form->getValues();
    $table = new Entries;
    $data = array(
        'title' => $values['title'],
        'date' => $values['date'],
        'author' => Zend_Auth::getInstance()->getIdentity()->username,
        'author_id' => Zend_Auth::getInstance()->getIdentity()->id,
        'body' => $values['entrybody'],
        'extended_body' => $values['entrybodyextended']
    );
    $table->insert($data);
    $this->view->entrySaved = true;
  }

  public function listAction()
  {
```

```
    $table = new Entries;
    $rows = $table->fetchAll(
        $table->select()->from($table, array('title','date','id'))
    );
    $this->view->entries = $rows->toArray();
}

public function editAction()
{
    $form = new ZFBlog_Form_EntryEdit;
    if (!$this->getRequest()->isPost()) {
        $table = new Entries;
        $rowset = $table->find( $this->_getParam('id') );
        if (count($rowset) == ) {
            $this->view->failedFind = true;
            return;
        }
        $form->setElementFilters(array()); // disable all element filters
        $this->_repopulateForm($form, $rowset->current());
        $this->view->entryForm = $form;
        return;
    } elseif (!$form->isValid($_POST)) {
        $this->view->failedValidation = true;
        $this->view->entryForm = $form;
        return;
    }

    $values = $form->getValues();
    $table = new Entries;
    $data = array(
        'title' => $values['title'],
        'date' => $values['date'],
        'body' => $values['entrybody'],
        'extended_body' => $values['entrybodyextended']
    );
    $table->update($data,
        $table->getAdapter()->quoteInto('id = ?', $values['id'])
    );
    $this->view->entrySaved = true;
}

public function deleteAction()
{

}

protected function _repopulateForm($form, $entry)
{
    $values = array(
        'title' => $this->_reverseAutoFormat($entry->title),
        'date' => $entry->date,
        'entrybody' => $this->_reverseAutoFormat($entry->body),
        'entrybodyextended' =>
```

```
            $this->_reverseAutoFormat($entry->extended_body),
        'id' => $this->_getParam('id')
    );
    $form->populate($values);
}

protected function _reverseAutoFormat($string)
{
    $string = preg_replace("/\/", '', $string);
    $string = preg_replace("/\/", "\n\n", $string);
    $string = preg_replace("/\/", '', $string);
    $string = preg_replace("/\/", '', $string);
    $string = html_entity_decode($string, ENT_QUOTES, 'UTF-8');
    return $string;
}
}
```

We're knee deep in complexity now... To keep the editAction() method cleaner, I've refactored a few steps into protected helper methods.

The workflow is quite straightforward. If no form is submitted, we populate a new form with reverse-filtered data so it's back in the same form we expect the author had originally inputted it. If we are dealing with a form submission, we simply check validity, and update the older entry on the database.

The main tricky bit is that _reverseAutoFormat() method. I'm certainly not comfortable with it since it makes a lot of assumptions - the worst one being to entity decode the data. It's a reasonable default, but chances are there will be some input the user expressly encoded for a reason. The content between `<code>` tags springs immediately to mind. In the future I would seriously consider a class expressly for reverse filtering that uses something more flexible like the DOM.

Here's the respective view template for our editAction():

```
<?php if($this->failedValidation): ?>
    <p class="error">Some problems were detected with the submitted form.</p>
<?php endif; ?>
<?php if($this->entrySaved): ?>
    <p class="success">Entry has been saved.</p>
<?php elseif($this->failedFind): ?>
    <p class="error">The entry could not be found in the database.</p>
<?php else: ?>
    <?php echo $this->entryForm ?>
<?php endif; ?>
```

The ever mounting feedback messages are becoming intolerable I hope?

Fire ahead, and try out adding and editing entries.

Conclusion

Blog entries. You can't live without them, and despite their apparent simplicity they can be complex to program for. We're only at the start of a chain of work that will see us building a layer of entry data processing possibilities. The main ones covered here is creating, cleaning and editing. At some point I'll want to parse out coloured PHP syntax, perhaps insert some Microformatting. Maybe throw in more autoformatting. The strategies governing these and converting between presentation forms and user editable forms are well worth a careful examination.

In the next Part to this series we'll select the simplest strategy of them all. We'll retain original input on the database, and simply post-process the entry for any such filtering. This will add heavily to page view performance but we do have options to limit that!

Note: The source code for this entry is available to browse, or checkout with subversion, from http://svn.astrumfutura.org/zfblog/tags/Part8. The full source code for the entire application (as it exists thus far) from http://svn.astrumfutura.org/zfblog.

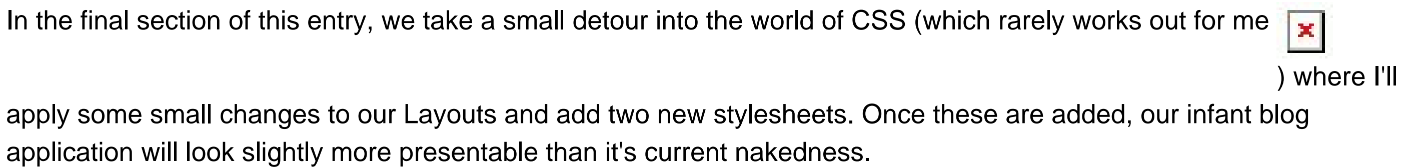
Posted by PÃ¡draic Brady in PHP General, PHP Security, Zend Framework at 19:39

Saturday, May 10. 2008

Example Zend Framework Blog Application Tutorial - Part 7: Authorisation with Zend_Acl and Revised Styling

You'd never think a guy could write so much about a blog application but to date after 6 parts we have covered a mass of detail from initial setup of our project's directory structure to Authentication of users. To date the feedback has been overwhelmingly positive to this series and I'm presently collecting comments regarding improvements for later inclusion.

Today's entry concerns authorisation. We previously covered how to authenticate an author to the blog, but we still have nothing ensuring only authenticated authors can access the new Administration Module. This is the domain of Zend_Acl, an implementation of an Access Control List system which limits access to resources by the roles assigned to a user.

In the final section of this entry, we take a small detour into the world of CSS (which rarely works out for me ) where I'll apply some small changes to our Layouts and add two new stylesheets. Once these are added, our infant blog application will look slightly more presentable than it's current nakedness.

Step 1: Understanding Access Control Lists (ACL)

It can be a bit confusing to face off against ACL if you're new to the subject. In essence all ACL does is keep track of resources and roles.

As to what a resource is, it is anything to which access can be allowed or denied. For our blog application, I could decide that the Administration Module is itself one resource. From there I can restrict all access to that entire Module, including all it's Controller classes and Action methods (which are part of that single Resource). Or perhaps I could determine that only one Action method in the whole Module is a specific Resource, bearing in mind that Resources are nestable (i.e. a basket is a Resource, and each egg it holds are also discrete Resources). Since each Resource can be given differing access rules, you can globally prevent non-author users from accessing the Administration Module, but maybe allow some registered users access to specific Actions in that Module as an exception to the global rule.

A lot of the time managing global rules, and then applying exception rules, is how ACL works in practice.

Explaining a Role is even simpler. Any visitor to the application can be assigned a Role which ACL rules may use to define that user's access to Resources. Typically the first Role everyone will receive is "guest". From there you can escalate Roles to offer a visitors a greater degree of access to Resources. Any user can be given multiple Roles even. For example, if an author visits the blog they start with the role of "guest" but after authentication we might grant them the additional role of "author". If Roles dictate specific but limited responsibilities (perhaps there's an "author" and "editor" Roles) you might decide to start tracking roles more elaborately, in a database possibly.

Going a bit further, if our Administration Module is a Resource called "admin" then we can decide that the only Role with access to it will be the "author" Role. Since our user has been authenticated and granted the "author" Role (either post-authentication or permanently recorded on the database), they can access the Administration Module.

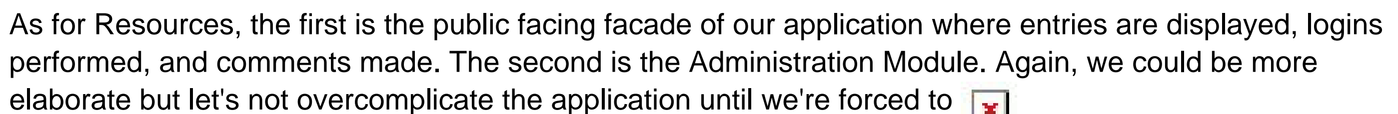
Finally is the concept of Privileges. Just because you can access a Resource, does not instantly mean you should have total uncontrolled access to it. You can limit control over a Resource using Privileges. Perhaps an Author can access the Admin Module (represented by an Admin Resource) but we want to deny Authors

the privilege of deleting entries from the database.

Step 2: A Little Planning Goes A Long Way

Before we leap into the fray like a demented action hero, let's set out exactly what we're aiming for.

Since our blog is a relatively simple application, we really only need two Roles to start with. We'll call these `guest` and `author`. This may change in the future, perhaps we could allow for multiple Authors but one Editor capable of editing all posts. In that case we'd need to pick apart how that's implemented. But for now, two Roles is just fine.

As for Resources, the first is the public facing facade of our application where entries are displayed, logins performed, and comments made. The second is the Administration Module. Again, we could be more elaborate but let's not overcomplicate the application until we're forced to  . This suggests we only have two Resources: the Default Module and the Administration Module. Remember that the Default Module comprises everything not assigned to a specific Module (like our Admin Module with its own separate tree of Controllers and Views).

The rules falling out form this quick analysis are simple.

1. Guests can access the Default Module
2. Authors can access the Default Module
3. Guests cannot access the Administration Module
4. Authors can access the Administration Module

Step 3: Storing Rules in a Class

There is no specific backend storage for Zend_Acl which since it is a serialisable class can be simply serialised and stored in anything from a database to a file for later consumption. To keep things simple I'll just implement a class defining the relationship between Roles and Resources as described above. You'll note we are not using Privileges, since access to a Resource assumes the accessing Role has all possible Privileges by default.

Start by creating a new file at `/library/ZFBlog/Acl.php`:

```
<?php
class ZFBlog_Acl extends Zend_Acl
{
  public function __construct(Zend_Auth $auth)
  {
    // Add Resources
    // Resource #1: Default Module
    $this->add(new Zend_Acl_Resource('default'));
    // Resource #2: Admin Module
    $this->add(new Zend_Acl_Resource('admin'));
    // Add Roles
    // Role #1: Guest
    $this->addRole(new Zend_Acl_Role('guest'));
    // Role #2: Author (inherits from Guest)
```

```
        $this->addRole(new Zend_Acl_Role('author'), 'guest');
        // Assign Access Rules
        // Rule #1 & #2: Guests can access Default Module (Author inherits this)
        $this->allow('guest', 'default');
        // Rule #3 & #4: Authors can access Admin Module (Guests denied by default)
        $this->allow('author', 'admin');
    }
}
```

One confusing point I've seen asked is whether Resources explicitly refer to Modules, Controllers or Actions. They don't - the names used here are pure convention. A resource is a virtual item. Zend_Acl doesn't know if a Resource is a Module, Controller or Action since that's a decision we make when we check the ACL rules later. What we'll do then is detect what Module a request for, and specifically carry out an ACL check for the Resource created here to refer to that Module, i.e. the connection between a real Module name and an ACL Virtual Resource is determined entirely by us at checking time.

Probably the biggest area of confusion is that its so common to consider Resources as Controllers, and Privileges as Actions, that people don't realise this is pure convention. Someone seeing `$this->allow('author', 'entry', array('create', 'edit', 'delete'))` would interpret that Authors are allowed access to the Entry Controller with privileges sufficient to create, edit or delete entries. That is only the case if, and when, your ACL logic determines this is how the rule is interpreted.

To prove a point, where do Modules fit? They don't. There is no Module parameter for `Zend_Acl::allow()` . You could append ACL interpretive logic where a Resource called "admin|entry" refers to the Entry Controller of an Admin Module which once again emphasises that a Resource is completely virtual. It is only what you interpret it to be and has no preconceived relationship with MVC. For all you care a Resource could be absolutely anything that is accessible only by passing through the ACL checkpoint.

Step 4: Implementing ACL using a custom Front Controller Plugin

The problem with ACL is that it's one of those always-on checks. Every single request needs to be checked to ensure that the requesting user has a Role which allows them to access the Resource (i.e. the Module, Controller or Action) being requested.

By stringing together the requirements (interacts with request data, operates on all requests, occurs prior to Controller execution) we realise that the best way of accomplishing this is to add a Front Controller plugin implementing the preDispatch() method.

Create a new file at `/library/ZFBlog/Controller/Plugin/Acl.php`.

Once again we're using the PEAR Convention and we will continue to mirror the organisation of Zend Framework classes.

```
<?php
class ZFBlog_Controller_Plugin_Acl extends Zend_Controller_Plugin_Abstract
{
    protected $_auth = null;
    protected $_acl = null;
    public function __construct(Zend_Auth $auth, Zend_Acl $acl)
    {
        $this->_auth = $auth;
```

```
        $this->_acl = $acl;
    }
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        // Before you lot start, this is the laziest possible
        // means of assigning roles. Hands up - I'm guilty!
        // Store to the Author table if you prefer.
        if ($this->_auth->hasIdentity()) {
            $role = 'author';
        } else {
            $role = 'guest';
        }

        // Mapping to determine which Resource the current
        // request refers to (really simple for this example!)
        $resource = $request->module;
        if (!$this->_acl->has($resource)) {
            $resource = null;
        }
        // ACL Access Check
        if (!$this->_acl->isAllowed($role, $resource)) {
            if ($this->_auth->hasIdentity()) {
                // authenticated, denied access, forward to index
                $request->setModuleName('default');
                $request->setControllerName('index');
                $request->setActionName('index');
            } else {
                // not authenticated, forward to login form
                $request->setModuleName('default');
                $request->setControllerName('author');
                $request->setActionName('login');
            }
        }
    }
}
```

Look Ma, it's another plugin!

If you haven't already guessed from reading this series so far, a lot of great things happen when you take the time to write plugins and helpers for the Zend Framework.

This Front Controller Plugin is really simple. It checks whether the current user is authenticated or not. If they are authenticated, it assumes they are an Author - not a Guest. Now assumptions are not very pretty, but it's a blog with two roles. If you had a slightly more complex scenario (say we had Guest, Author and Editor roles) you could instead maintain Roles in the database for querying by this plugin. For example, add a new "role" field to our author table on the database - the data from it will get stored by Zend_Auth during authentication which, as you see, is accessible by this plugin .

After Roles are assumed, we grab the name of the current Module and since we have a convention that Resource names map one-for-one to Module names, it's a simple check to see if the current user's Role

allows them access to that Resource. If it does, we simply do nothing and let control pass back to the Front Controller. If access is denied we have two branches - authenticated users are just kicked back to the index page (part of our default Module) while unauthenticated users are redirected to the login page for Authors.

Note: Setting the Module name on the request object for forwarding is required when using Modules. That includes setting references to non-moduled controllers/actions with the Module name of "default".

Of course there are more complex scenarios again. What if the resource is not a Module, but a specific Action on a specific Controller within a Module? Obviously such a simple plugin as above would fall flat and need to be scraped off the floor. Again this is not dictated to you by the Zend Framework manual. It's usually typical to make use of Privileges in Resources and interpret these as Actions. And as explained earlier you can use a "admin|entry" convention for the Module/Controller pairing.

Step 5: Initialising the Front Controller Plugin

Before our plugin is even used, we need to register it with the Front Controller. This is another change to our Bootstrap class! The new method `setupAcl` is at the bottom of the file.

```
<?php
require_once 'Zend/Loader.php';
class Bootstrap
{
   public static $frontController = null;
   public static $root = '';
   public static $registry = null;
   public static function run()
   {
      self::prepare();
      $response = self::$frontController->dispatch();
      self::sendResponse($response);
   }
   public static function setupEnvironment()
   {
      error_reporting(E_ALL|E_STRICT);
      ini_set('display_errors', true);
      date_default_timezone_set('Europe/London');
      self::$root = dirname(dirname(<u>_FILE_</u>));
   }
   public static function prepare()
   {
      self::setupEnvironment();
      Zend_Loader::registerAutoload();
      self::setupRegistry();
      self::setupConfiguration();
      self::setupFrontController();
      self::setupView();
      self::setupDatabase();
      self::setupAcl();
   }
   public static function setupFrontController()
   {
      self::$frontController = Zend_Controller_Front::getInstance();
```

```
    self::$frontController->throwExceptions(true);
    self::$frontController->returnResponse(true);
    self::$frontController->setControllerDirectory(
        array(
            'default' => self::$root . '/application/controllers',
            'admin' => self::$root . '/application/admin/controllers'
        )
    );
    self::$frontController->setParam('registry', self::$registry);
}
public static function setupView()
{
    $view = new Zend_View;
    $view->setEncoding('UTF-8');
    $viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
    Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
    Zend_Layout::startMvc(
        array(
            'layoutPath' => self::$root . '/application/views/layouts',
            'layout' => 'common',
            'pluginClass' => 'ZFBlog_Layout_Controller_Plugin_Layout'
        )
    );
}
public static function sendResponse(Zend_Controller_Response_Http $response)
{
    $response->setHeader('Content-Type', 'text/html; charset=UTF-8', true);
    $response->sendResponse();
}
public static function setupRegistry()
{
    self::$registry = new Zend_Registry(array(), ArrayObject::ARRAY_AS_PROPS);
    Zend_Registry::setInstance(self::$registry);
}
public static function setupConfiguration()
{
    $config = new Zend_Config_Ini(
        self::$root . '/config/config.ini',
        'general'
    );
    self::$registry->configuration = $config;
}
public static function setupDatabase()
{
    $config = self::$registry->configuration;
    $db = Zend_Db::factory($config->db->adapter, $config->db->toArray());
    $db->query("SET NAMES 'utf8'");
    self::$registry->database = $db;
    Zend_Db_Table::setDefaultAdapter($db);
}
public static function setupAcl()
{
    $auth = Zend_Auth::getInstance();
```

```
        $acl = new ZFBlog_Acl($auth);
        self::$frontController->setParam('auth', $auth);
        self::$frontController->setParam('acl', $acl);
        self::$frontController->registerPlugin(
          new ZFBlog_Controller_Plugin_Acl($auth, $acl)
        );
    }
}
```

In case they are needed for a more specific use case in a controller, I've added both the authentication and authorisation objects as Front Controller parameters.

Go boot up `http://zfblog/admin` for a test drive. If you have previously logged in, you can logout using `http://zfblog/author/logout` manually (it's not part of our View yet).

Step 6: And Now For Something Completely Different...

With ACL implemented I find myself at a loose end. So I spent some time putting together two new stylesheets to add some colour and life to this application. You'll need two new files:

```
/public/css/style.css
/public/css/ie.css
```

The first may already exist from an earlier Part of this series.

Edit `style.css` to contain the following CSS. I won't explain it - this, thankfully, is not a CSS blog .

```
body {
  margin: ;
}
#content {
  min-height: 50em;
}
#header {
      background: #303030;
}
#header h1 {
      float: left;
      width: 235px;
      height: 80px;
      margin: ;
}
#header a {
  color: #AAA;
  text-decoration: none;
}
#footer {
  clear: both;
  width: 100%;
```

```
    margin: ;
    padding: 15px ;
    border-top: 1px solid #000;
    background: #303030;
    text-align: center;
    color: #AAA;
}
fieldset {
    float: left;
    clear: both;
    width: 100%;
    margin:   1.5em ;
    padding: ;
    border: 1px solid #BFBAB0;
    background-color: #F2EFE9;
}
fieldset.submit {
    float: none;
    width: auto;
    border-style: none;
    padding-left: 13.5em;
    background-color: transparent;
}
legend {
    margin-left: 1em;
    padding: ;
    color: #000;
    font-weight: bold;
}
fieldset ol {
    padding: 1em 1em  1em;
    list-style: none;
}
fieldset li {
    float: left;
    clear: left;
    width: 100%;
    padding-bottom: 1em;
}
label {
    float: left;
    width: 10em;
    margin-right: 1em;
    text-align: right;
}
label strong {
    display: block;
    color: #C00;
    font-size: 85%;
    font-weight: normal;
    text-transform: uppercase;
}
label em {
```

```
    display: block;
    color: #060;
    font-size: 85%;
    font-style: normal;
    text-transform: uppercase;
}
```

Now edit `ie.css`.

```
#content {
    height:auto !important;
    height:50em;
}
legend {
    position: relative;
    top: 7px;
}
fieldset {
    margin-top: 2em;
    margin-bottom: ;
    position: relative;
}
fieldset ol {
    padding: ;
}
fieldset.submit {
    margin-bottom: 1.5em;
    padding-left: 13.2em;
}
```

Now repeat after me: "Paddy is not a designer!". I can't say this is the most spectacular CSS ever written when it's probably more in the opposite direction. But it works. I'm pretty sure it does anyway. Fingers crossed! There might be some IE7 issues because my conditional doesn't specify a version so check back with subversion in a while if so.

We're not done yet. We need to make two more edits to each of our previous Layout templates. Here's the amended excerpt for each. It is identical for both `/application/views/layouts/common.phtml` and `/application/admin/views/layouts/admin.phtml`. Now since it's identical we know we have some duplication in our layouts, which is undesirable, but we'll fix that another day.

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <meta name="language" content="en" />
    <title><?php echo $this->escape($this->title) ?></title>
    <link rel="stylesheet" href="/css/blueprint/screen.css" type="text/css" media="screen, projection"
>
    <link rel="stylesheet" href="/css/style.css" type="text/css" media="screen, projection">
    <link rel="stylesheet" href="/css/blueprint/print.css" type="text/css" media="print">
    <!--[if IE]>
    <link rel="stylesheet" href="/css/blueprint/ie.css" type="text/css" media="screen, projection">
    <link rel="stylesheet" href="/css/ie.css" type="text/css" media="screen, projection">
```

```
    <![endif]-->
</head>
<body>

    <div class="container">
        <div class="block">
            <div id="header" class="column span-24">
                <h1><a href="/">Lorem Ipsum</a></h1>
            </div>
        </div>

// ...............
        <div class="block">
            <div id="footer" class="span-24">
                <p>Copyright &copy; 2008 PÃ¡draic Brady</p>
            </div>
        </div>
    </div>
</body>
</html>
```

Just make matters worse, I added some small changes to our previous AuthorController and it's related login template.

/application/controllers/AuthorController.php

```
<?php
class AuthorController extends Zend_Controller_Action
{
    public function loginAction()
    {
        $form = new ZFBlog_Form_AuthorLogin;
        if (!$this->getRequest()->isPost()) {
            $this->view->loginForm = $form;
            return;
        } elseif (!$form->isValid($_POST)) {
            $this->view->failedValidation = true;
            $this->view->loginForm = $form;
            return;
        }
        $values = $form->getValues();
        // Setup DbTable adapter
        $adapter = new Zend_Auth_Adapter_DbTable(
            Zend_Db_Table::getDefaultAdapter() // set earlier in Bootstrap
        );
        $adapter->setTableName('authors');
        $adapter->setIdentityColumn('username');
        $adapter->setCredentialColumn('password');
        $adapter->setIdentity($values['name']);
        $adapter->setCredential(
            hash('SHA256', $values['password'])
        );
        // authentication attempt
```

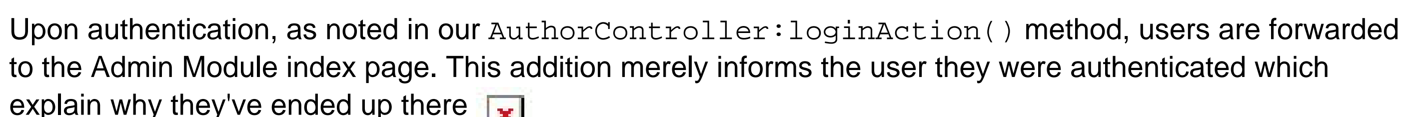
```
        $auth = Zend_Auth::getInstance();
        $result = $auth->authenticate($adapter);
        // authentication succeeded
        if ($result->isValid()) {
            $auth->getStorage()
                ->write($adapter->getResultRowObject(null, 'password'));
            $this->view->passedAuthentication = true;
            $this->_forward('index', 'index', 'admin');
        } else { // or not! Back to the login page!
            $this->view->failedAuthentication = true;
            $this->view->loginForm = $form;
        }
    }
    public function logoutAction()
    {
        Zend_Auth::getInstance()->clearIdentity();
        $this->_helper->redirector('index', 'index');
    }
}
```

The main change here was to add additional View variables to flag some conditions to the view template as used below. I tend to use flags a lot with Views since they are simple booleans and delegate the translation of these flags into some meaning to the View itself. It's a really obvious gimmick I know but there are still lots of people who assign actual text without a second thought.

```
/application/views/scripts/author/login.phtml
<h2>Authentication</h2>
<p>Enter your author name and password below.</p>
<?php if($this->failedValidation): ?>
    <p class="error">Some problems were detected with the submitted form.</p>
<?php elseif($this->failedAuthentication): ?>
    <p class="error">The credentials supplied could not be authenticated. Please try again.</p>
<?php endif; ?>
<?php echo $this->loginForm->render() ?>
```

Lastly I made another tiny change to the template for our Administration Modules index page:

```
/application/admin/views/scripts/index/index.phtml
<h2>Administration</h2>
<?php if($this->passedAuthentication): ?>
    <p class="success">You have been successfully authenticated.</p>
<?php endif; ?>
<ul>
<li><a href="/admin/entry/add">New Entry</a></li>
</ul>
```

Upon authentication, as noted in our `AuthorController:loginAction()` method, users are forwarded to the Admin Module index page. This addition merely informs the user they were authenticated which explain why they've ended up there 

.

Step 7: A Small Administration Template Update

Hindsight is great, but unfortunately you can't make use of it on a live blog, so I'll make a few quick changes now. What we'll do is create a simple Administration Menu in the left column of our Administration Module's templates.

Open up the file `/application/admin/views/scripts/index/index.phtml` and edit as follows:

```
<h2>Administration</h2>
<?php if($this->passedAuthentication): ?>
  <p class="success">You have been successfully authenticated.</p>
<?php endif; ?>
<p>Welcome, <?php echo Zend_Auth::getInstance()->getIdentity()->realname ?>.</p>
```

The change merely replaces the early menu block with a simple greeting using the Author's real name as stored from previous authentication.

Open up the file `/application/admin/views/layouts/admin.phtml` and edit as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <meta name="language" content="en" />
  <title><?php echo $this->escape($this->title) ?></title>
  <link rel="stylesheet" href="/css/blueprint/screen.css" type="text/css" media="screen, projection"
>
  <link rel="stylesheet" href="/css/style.css" type="text/css" media="screen, projection">
  <link rel="stylesheet" href="/css/blueprint/print.css" type="text/css" media="print">
  <!--[if IE]>
  <link rel="stylesheet" href="/css/blueprint/ie.css" type="text/css" media="screen, projection">
  <link rel="stylesheet" href="/css/ie.css" type="text/css" media="screen, projection">
  <![endif]-->
</head>
<body>

  <div class="container">
    <div class="block">
      <div id="header" class="column span-24">
        <h1><a href="/">Lorem Ipsum</a></h1>
      </div>
    </div>
    <div class="block">
      <div id="left" class="column span-5">
        <div class="menu">
          <ul>
            <li class="menu-header">Account</li>
            <li><a href="/author/logout">Logout</a></li>
```

```
                <li class="menu-header">Entries</li>
                <li><a href="/admin/entry/add">New Entry</a></li>
            </ul>
        </div>
    </div>

    <div id="content" class="column span-18">

        <?php echo $this->layout()->content ?>
    </div>
</div>
<div class="block">
    <div id="footer" class="span-24">
        <p>Copyright &copy; 2008 PÃ¡draic Brady</p>
    </div>
</div>
</div>
</body>
</html>
```

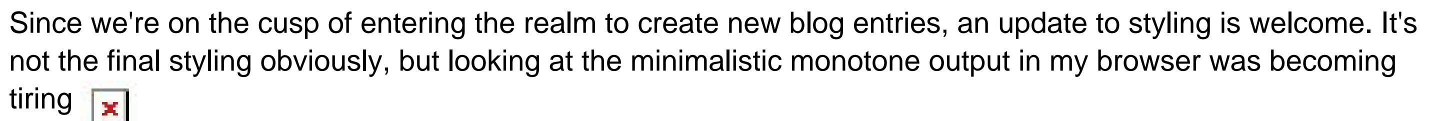
If editing the menu in looks odd, don't worry yet. Those who really don't like it can add it as a template partial or pass an array of menu data into a custom View Helper (I'll be doing something similar in the near future to tidy up layouts). Finally, edit our stylesheet at `/public/css/style.css` to include the following styles for the now modified column.

```
div.menu {
    padding:  1em  1em;
    margin-top: 3em;
}
div.menu ul {
    list-style: none;
}
div.menu ul li.menu-header {
    padding-top: 1em;
    font-weight: bold;
    font-size: 110%;
}
div.menu ul > li:first-child.menu-header {
    padding-top: ;
}
div.menu a {
    text-decoration: none;
    font-weight: bold;
}
```

Try out the application once more. It should now look a bit more presentable.

Conclusion

Implementing an ACL solution with Zend_Acl hasn't proven difficult, in part due to the simplicity of our immediate needs. I would reiterate that remembering that Resources are not Controllers/Action, but merely virtual values which can be mapped or interpreted as such, is quite important. Using a 1:1 mapping is simply a common convention.

Since we're on the cusp of entering the realm to create new blog entries, an update to styling is welcome. It's not the final styling obviously, but looking at the minimalistic monotone output in my browser was becoming tiring 
.

In the next edition of this series we'll visit the task of creating, editing and displaying new blog entries.
Posted by PÃ¡draic Brady in PHP General, PHP Security, Zend Framework at 00:48

Thursday, May 8. 2008

Example Zend Framework Blog Application Tutorial - Part 6: Introduction to Zend_Form and Authentication with Zend_Auth

In the previous entry, we created a new Administration Module to hold blog management functionality, added a Module specific layout for it, and discussed the upcoming need to ensure this is only accessible by authorised Authors. In this entry I'll unravel some of Zend_Form's mysteries in adding a login form, before using Zend_Auth to implement authentication for authors.

Previously: Part 5: Creating Models with Zend_Db and adding an Administration Module

Authentication in the Zend Framework is the domain of the Zend_Auth component, and it is really easy to use. Zend_Auth is really an abstract API to a number of components working in concert, and without the usual micromanagement of database interaction, sessions, cookies and user data persistence, it makes my life a lot simpler. Of course authentication demands a login form, and so I'll first visit using Zend_Form. Zend_Form is an interesting component because it's one of the worst to get started with. The manual, as it does for all components, does not impose a best practice to setting up forms. Mix that with the number of form organisations possible (class based, config based, view template based) and it can be very confusing.

Step 1: Adding a Login Action and View

Before we actually perform authentication, we need a login form. I've decided to attach all Author account actions to an Author Controller. Add a new file called `AuthorController.php` in `/application/controllers/` containing the following:

```
<?php
class AuthorController extends Zend_Controller_Action
{

    public function loginAction()
    {
    }
    public function logoutAction()
    {
        $this->_forward('index', 'index');
    }

}
```

The logout action for the moment does nothing, but forwards `/author/logout` requests to the main index, just as I would intend to occur after a real logout.

We'll also add a matching template at `/application/views/scripts/author/login.phtml`:

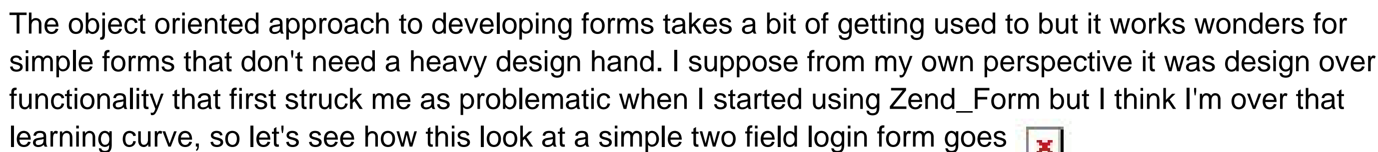
```
<h2>Authentication</h2>
<p>Enter your author name and password below.</p>
<?php echo $this->loginForm ?>
```

Nothing major here, except for a mysterious reference to a view variable, $loginForm!

Step 2: Creating a Login form with Zend_Form

Zend_Form is one of the most recent additions to the Zend Framework with the release of 1.5. It's not surprising it took so long since a decent Form library is not a trivial component to get through development.

The object oriented approach to developing forms takes a bit of getting used to but it works wonders for simple forms that don't need a heavy design hand. I suppose from my own perspective it was design over functionality that first struck me as problematic when I started using Zend_Form but I think I'm over that learning curve, so let's see how this look at a simple two field login form goes

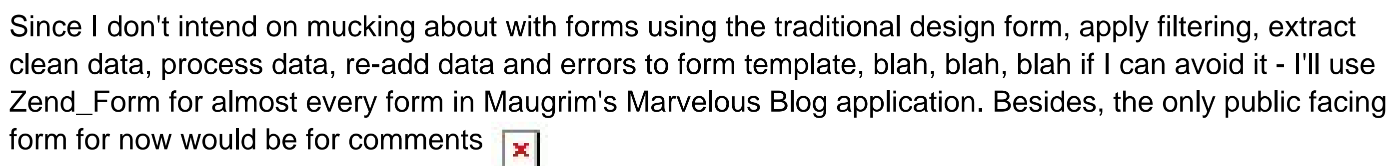


.

I've deliberately selected a preferred form style to adhere to so this will necessitate customising Zend_Form options and decorators. It's standard based, tableless, composed of semantic markup, and still looks okay without CSS styling or when using a screenreader (which is one of the more important facets for a form im my opinion).

Like a lot of areas in the Zend Framework, actually organising Form objects is left to your imagination. My first question when approaching any potential object nearly always concerns how reusable I can make it. A reusable form object assumes I'll end up implementing a standard subclass of Zend_Form so I don't have to repeat myself a dozen times in concrete classes. Hopefully this section provides you with a few good ideas - I have seen Zend_Form examples in the wild that are horrific so I will spend a chunk of time on Zend_Form on this outing.

Since I don't intend on mucking about with forms using the traditional design form, apply filtering, extract clean data, process data, re-add data and errors to form template, blah, blah, blah if I can avoid it - I'll use Zend_Form for almost every form in Maugrim's Marvelous Blog application. Besides, the only public facing form for now would be for comments



.

Here's the proposed output I'd be seeking with all this:

```
<form action="/author/login">
<fieldset>
  <legend>Author Authentication</legend>
  <ol>
    <li>
      <label for="name">Name:</label>
      <input type="text" name="name" id="name" />
    </li>
    <li>
      <label for="password">Password:</label>
      <input type="password" name="password" id="password" />
    </li>
  </ol>
</fieldset>
<fieldset class="form-button">
  <input type="submit" value="Submit" />
</fieldset>
</form>
```

Let's see if I can kick Zend_Form into cooperating with me on generating that! Or something similar at least...

If Zend_Form has a flaw, the biggest one is its documentation because like most of the Zend components it doesn't dictate a best practice for organising the final classes. It also appears a little vague at times, but still it does answer a lot of questions if you read it attentive to detail. If something here is just not making sense do ask in the comments or on the Zend Framework mailing lists.

My own take is to take the term "Divide and Conquer" as my motto for dealing with Zend_Form. Break down each specific stage, and from there dump each stage into its own class family. Tackling the whole thing up front without some groundwork is a recipe for the most unmaintainable ugly looking Zend_Form implementation imaginable.

With forms our divisions to conquer are quite simple to visualise. We have the form elements which are segregated from any hint of presentation and which carry elements of business logic by virtue of them containing validation/filtering logic. We have the form element decoration which surrounds form elements with semantic markup styled by CSS we can write independently. Finally we have overall layout which groups all elements logically.

At it's simplest, this suggests we'll have two sets of classes. One group for form elements, and another for decorating those form elements. Take the suggested markup from earlier. The `li` tags are obviously decorators of the form elements they wrap. The `fieldset` tags pose another difficultly in that they are not specific to a form element group (there are 2 of them, and the placement is purely for presentation) and so we need some sort of form element grouping mechanism to apply decorators to.

The Zend_Form component does precisely that. Despite any confusion it may cause you, it does have a very logical setup for grouping and decorating elements.

Decorators will cause you a few headaches but they aren't completely nuts . The four main standard types are `ViewHelper`, `Label`, `HtmlTag` and `Error`. There are fourteen in total for even more decorating madness. By default, each of these effects the form element by adding to it's markup in a predefined (but usually configurable) way.

For example, you can use `Label` to prepend a form element with some markup. Obviously the name suggests a `label` tag for the form element but any tag is allowable. Or you can use `HtmlTag` to wrap tags around a form element like a set of `li` tags. Hit the manual for a full description of all 14 decorators.

Revisiting our markup, we can suggest a few decorator steps, by working our way out from the form element. The more specific we get here the better.

1. Prepend form element with `label` tags
2. Wrap form element and `label` with `li` tags
3. Prevent application of 1 and 2 to `submit` element
4. Eliminate default decoration of `submit` element

Once we hit this point in the decorator flow, we leave the domain of the individual form element. `ol` and `fieldset` decorate groups of form elements. Zend_Form wins again by letting us create Display Groups for decorating groups of form elements.

5. Create display group for each `fieldset`
6. Wrap display group (name, password) with `ol` tags
7. Prepend display group (name, password) with `legend` tags

8. Wrap display group (name, password) with `fieldset` tags
9. Wrap display group (submit) with `fieldset` tags

Hopefully you're getting the picture. Pick apart your form, and take it step by step. There is a very logical flow with Zend_Form that makes implementing each step simple once you identify those steps! If you just jump in you will get lost unless you're already very comfortable with Zend_Form.

It's again really important that you watch the ordering of decorators - the first decorator added to the stack will be the first applied to the bare bones form element. So each step should be implemented from the inside out in order.

Delving back into PHP, we can now encode a few standard decorator arrays in a Zend_Form subclass. We do need to remember we have one exception above - the `submit` element requires special treatment since it's not decorated as an unordered list. In this subclass we'll also set a few defaults for all forms such as setting a new "accept-charset" attribute for the form element, and ensuring the rendered form object is free of any default decoration apart from the wrapping `form` element itself obviously.

Create the file `/library/ZFBlog/Form.php` as part of our slowly growing application specific library.

```php
<?php
class ZFBlog_Form extends Zend_Form
{
   protected $_standardElementDecorator = array(
      'ViewHelper',
      array('Label'),
      array('HtmlTag', array('tag'=>'li'))
   );
   protected $_buttonElementDecorator = array(
      'ViewHelper'
   );
   protected $_standardGroupDecorator = array(
      'FormElements',
      array('HtmlTag', array('tag'=>'ol')),
      'Fieldset'
   );
   protected $_buttonGroupDecorator = array(
      'FormElements',
      'Fieldset'
   );
   public function __construct($options = null)
   {
      parent::__construct($options);
      $this->setAttrib('accept-charset', 'UTF-8');
      $this->setDecorators(array(
         'FormElements',
         'Form'
      ));
   }
}
```

The decorator arrays defined as protected properties are quite simple. For example the `standardGroupDecorator` wraps a display group with the HTML `ol` tag, and then wraps this again with a

`fieldset` tag. The "FormElements" decorator is sort of a group dynamic - we want to render these group HTML tags around all the form elements included in this display group. You'll see this decorator used for any decoration of a display group of form elements, including the parent form as a single display block (helps to think of a form as a single parent display group with children).

As another example, the `standardElementDecorator` prepends a form element with a `label` (notably the label content is not defined yet), and wraps both the form element and label in `li` tags. You'll notice the standard group decorator wraps the parent `ol` tags around all of this.

See how easy that was? Once you break down your form markup into a series of specific decoration steps, writing Zend_Form decorator arrays tailored to those steps is pretty simple.

So we have a platform for making forms. How about something specific - like a Login form?

Create a new file `/library/ZFBlog/Form/AuthorLogin.php`:

```
<?php
class ZFBlog_Form_AuthorLogin extends ZFBlog_Form
{
   public function init()
   {
      $this->setAction('/author/login');
      // Display Group #1 : Credentials
      $this->addElement('text', 'name', array(
         'decorators' => $this->_standardElementDecorator,
         'label' => 'Name:'
      ));
      $this->addElement('password', 'password', array(
         'decorators' => $this->_standardElementDecorator,
         'label' => 'Password:'
      ));
      $this->addDisplayGroup(
         array('name', 'password'), 'authorlogin',
         array(
            'disableLoadDefaultDecorators' => true,
            'decorators' => $this->_standardGroupDecorator,
            'legend' => 'Credentials'
         )
      );
      // Display Group #2 : Submit
      $this->addElement('submit', 'submit', array(
         'decorators' => $this->_buttonElementDecorator,
         'label' => 'Submit'
      ));
      $this->addDisplayGroup(
         array('submit'), 'authorloginsubmit',
         array(
            'disableLoadDefaultDecorators' => true,
            'decorators' => $this->_buttonGroupDecorator
         )
      );
   }
}
```

How's that for something interesting? Now all we do for an author login form is define the elements, assign a standard or specific decorator array from the parent class, and assign to a display group which also gets a decorator array from the parent class assigned.

The only real sore point here perhaps, is that I'm declaring the text content of labels and legends in the source code. Some bright mind might find it better to stuff these into a configuration file, and maybe link up a Translator for good measure (not covered here but is a documented possibility in the manual), but for now it's enough to work with.

Let's now revise our AuthorController to create this form and pass it to the View.

```php
<?php
class AuthorController extends Zend_Controller_Action
{

    public function loginAction()
    {
        $form = new ZFBlog_Form_AuthorLogin;
        $this->view->loginForm = $form;
    }
    public function logoutAction()
    {
    }

}
```

Go ahead and open up your browser to `http://zfblog/author/login`.

Here's the exact output I get from Firefox without tinkering with element separator options:

```html
<form enctype="application/x-www-form-urlencoded" action="/author/login" accept-charset="UTF-8" method="post">
<fieldset id="authorlogin"><legend>Credentials</legend>
<ol>
<li><label for="name" class="optional">Name:</label>
<input type="text" name="name" id="name" value=""></li>
<li><label for="password" class="optional">Password:</label>
<input type="password" name="password" id="password" value=""></li></ol></fieldset>
<fieldset id="authorloginsubmit">
<input type="submit" name="submit" id="submit" value="Submit"></fieldset></form>
```

Apart from needing a few newline separators, it's pretty much what I set out to create . Zend_Form just added a few default classes and attribute values.

Step 3: Adding Validation to Login Form

Zend_Form isn't just for presentation since you can also make forms self validating. This takes advantage of all the standard validators you might expect to use manually. Let's revisit the ZFBlog_Form_AuthorLogin form

class after including some validation rules.

```
<?php
class ZFBlog_Form_AuthorLogin extends ZFBlog_Form
{
    public function init()
    {
        $this->setAction('/author/login');
        // Display Group #1 : Credentials
        $this->addElement('text', 'name', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Name:',
            'validators' => array(
                array('StringLength', false, array(5,20))
            ),
            'required' => true
        ));
        $this->addElement('password', 'password', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Password:',
            'required' => true
        ));
        $this->addDisplayGroup(
            array('name', 'password'), 'authorlogin',
            array(
                'disableLoadDefaultDecorators' => true,
                'decorators' => $this->_standardGroupDecorator,
                'legend' => 'Credentials'
            )
        );
        // Display Group #2 : Submit
        $this->addElement('submit', 'submit', array(
            'decorators' => $this->_buttonElementDecorator,
            'label' => 'Submit'
        ));
        $this->addDisplayGroup(
            array('submit'), 'authorloginsubmit',
            array(
                'disableLoadDefaultDecorators' => true,
                'decorators' => $this->_buttonGroupDecorator,
                'class' => 'submit' // fieldset class attribute for some later styling
            )
        );
    }
}
```

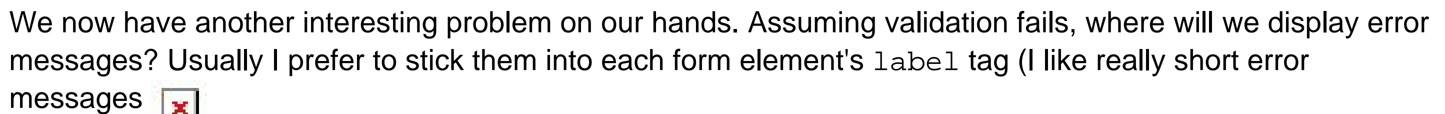
Note the additions including a new "required" flag dictating these values are required (i.e. must not be empty) and also a "validator" array giving a string length minimum and maximum for the "name" form element. I also just threw in a class attribute for our `authorloginsubmit` fieldset display group - it'll make styling that fieldset easier.

One thing blatantly missing is error messages - we haven't added any decorators to our form elements to

include error message text somewhere.

Step 4: Handling Error Messages with a Custom Decorator

We now have another interesting problem on our hands. Assuming validation fails, where will we display error messages? Usually I prefer to stick them into each form element's `label` tag (I like really short error messages



) whereas by default, remembering that we've now either disabled or overridden Zend_Form's decoration defaults, it's displayed separately by an appending `Error` decorator.

Since we're departing from the norm, it's time to customise how labels are generated.

What we need to do is intercept a label before it's rendered, and append error messages to it's content. Simplest way of doing that is to subclass the Label decorator. There is one additional thing to watch out for which is the fact that the standard Label decorator (when used to generate a `label` element) makes use of the FormLabel View Helper in Zend_View. If we intend appending HTML to the label name (as here) we'll need to disable the FormLabel Helper's default treatment of escaping the label name.

This points out another interesting nugget to remember. Many form decorators use Zend_View's View Helpers. And any options passed to a decorator, are also made available to the relevant View Helper. So it does pay to know the View Helpers as well as Zend_Form - the helpers have another layer of configurable behaviour you might find handy.

About that custom decorator, it's very simple. It just grabs the error messages for any element it's decorating and appends them to the label name wrapped in `strong` tags for styling access with CSS. Create a new file for the decorator at `/library/ZFBlog/Form/Decorator/LabelError.php`:

```php
<?php
class ZFBlog_Form_Decorator_LabelError extends Zend_Form_Decorator_Label
{
   public function getLabel()
   {
      $element = $this->getElement();
      $errors = $element->getMessages();
      if (empty($errors)) {
         return parent::getLabel();
      }
      $label = trim($element->getLabel());
      $label .= ' <strong>'
         . implode('</strong><br /><strong>', $errors)
         . '</strong>';
      $element->setLabel($label);
      return parent::getLabel();
   }
}
```

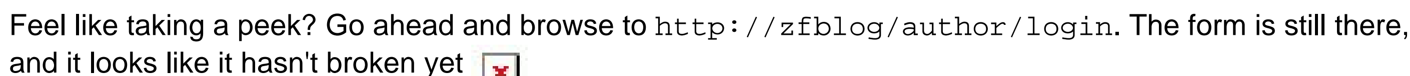
Here we're subclassing the `getLabel()` method - easier this way than retyping the whole original `render()` method! All the method does is append the error messages - we refer back to the parent class version of this method afterwards so we don't duplicate any source code from the standard decorator we're extending.

This is great - so let's wrap up by amending our `ZFBlog_Form` class to tell Zend_Form where to find our custom decorator (and indeed any future ones), and also add the new `LabelError` decorator to one of our decorator stacks.

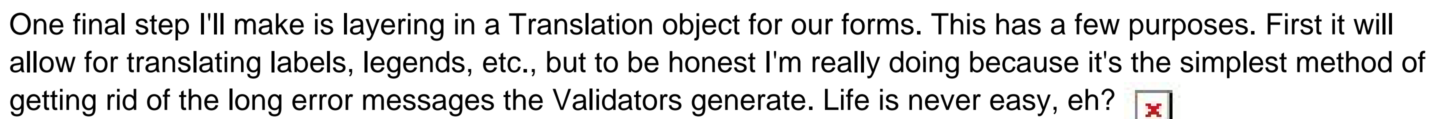
```php
<?php
class ZFBlog_Form extends Zend_Form
{
    protected $_standardElementDecorator = array(
        'ViewHelper',
        array('LabelError', array('escape'=>false)),
        array('HtmlTag', array('tag'=>'li'))
    );
    protected $_buttonElementDecorator = array(
        'ViewHelper'
    );
    protected $_standardGroupDecorator = array(
        'FormElements',
        array('HtmlTag', array('tag'=>'ol')),
        'Fieldset'
    );
    protected $_buttonGroupDecorator = array(
        'FormElements',
        'Fieldset'
    );
    public function __construct($options = null)
    {
        // Path setting for custom decorations MUST ALWAYS be first!
        $this->addElementPrefixPath('ZFBlog_Form_Decorator', 'ZFBlog/Form/Decorator/', 'decorator');
        parent::__construct($options);
        $this->setAttrib('accept-charset', 'UTF-8');
        $this->setDecorators(array(
            'FormElements',
            'Form'
        ));
    }
}
```

The new decorator reference receives a new option - we disable escaping of the label's name value so any included HTML doesn't get the htmlspecialchars treatment. This option is passed into the `FormLabel` View Helper when called from the decorator class.

In our constructor we have a comment. Never, ever, ever, forget that comment. It is essential that decorator paths for your custom decorators are added as early as possible - in fact they should be the first thing you do in a subclass of Zend_Form even before passing options to the parent's constructor.

Feel like taking a peek? Go ahead and browse to `http://zfblog/author/login`. The form is still there, and it looks like it hasn't broken yet 
. Go team!

Step 5: Replacing Those Cumbersome Default Errors

One final step I'll make is layering in a Translation object for our forms. This has a few purposes. First it will allow for translating labels, legends, etc., but to be honest I'm really doing because it's the simplest method of getting rid of the long error messages the Validators generate. Life is never easy, eh? 

We'll start simple and only address error messages as a quick example. Add a new file at `/translate/forms.php` containing:

```
<?php
return array(
   Zend_Validate_NotEmpty::IS_EMPTY => 'Required',
   Zend_Validate_StringLength::TOO_SHORT => 'Minimum Length of %min%',
   Zend_Validate_StringLength::TOO_LONG => 'Maximum Length of %max%'
);
```

With the translation file in place (using the Zend_Translate Array adapter) we just need to tell Zend_Form where to find it. Zend_Form uses a static method for this, so I'll create a quick static check and helper function for our `ZFBlog_Form` class.

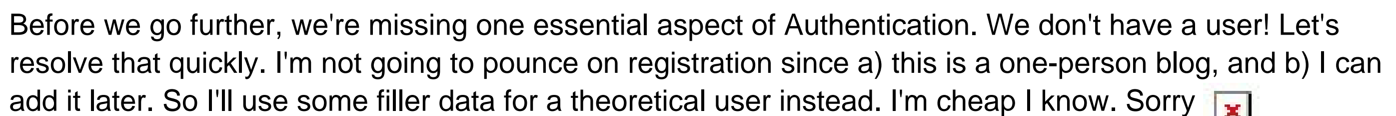
```
<?php
class ZFBlog_Form extends Zend_Form
{
   protected $_standardElementDecorator = array(
      'ViewHelper',
      array('LabelError', array('escape'=>false)),
      array('HtmlTag', array('tag'=>'li'))
   );
   protected $_buttonElementDecorator = array(
      'ViewHelper'
   );
   protected $_standardGroupDecorator = array(
      'FormElements',
      array('HtmlTag', array('tag'=>'ol')),
      'Fieldset'
   );
   protected $_buttonGroupDecorator = array(
      'FormElements',
      'Fieldset'
   );
   public function __construct($options = null)
   {
      // Path setting for custom decorations MUST ALWAYS be first!
      $this->addElementPrefixPath('ZFBlog_Form_Decorator', 'ZFBlog/Form/Decorator/', 'decorator');
      $this->_setupTranslation();
      parent::__construct($options);
      $this->setAttrib('accept-charset', 'UTF-8');
      $this->setDecorators(array(
         'FormElements',
         'Form'
      ));
   }
```

```
    protected function _setupTranslation()
    {
        if (self::getDefaultTranslator()) {
            return;
        }
        $path = dirname(dirname(dirname(<u>_FILE_</u>)))
            . '/translate/forms.php';
        $translate = new Zend_Translate('array', $path, 'en');
        Zend_Form::setDefaultTranslator($translate);
    }
}
```

Easy as pie. Now all error messages will be replaced.

Step 6: Introducing Joe Bloggs

Before we go further, we're missing one essential aspect of Authentication. We don't have a user! Let's resolve that quickly. I'm not going to pounce on registration since a) this is a one-person blog, and b) I can add it later. So I'll use some filler data for a theoretical user instead. I'm cheap I know. Sorry



.

Here's Joe. Rumour has it he's The Common Man (TM). If you prefer, Joe can be a Jane. Nobody intends hunting him/her down to check anyway...

```
INSERT INTO `authors` (`realname`, `username`, `password`, `email`) VALUES ('Joe
Bloggs', 'joebloggs',
'5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8',
'joe.bloggs@example.com');
```

The hashed password is "password" hashed using the SHA256 algorithm.

Run the SQL insert on your users table in the database to manually register Joe (or Jane).

Step 7: Adding Authentication and Form Processing to AuthorController

Zend_Auth operates by referencing an adapter for our storage system. Since we're using a database, we'll use the `Zend_Auth_Adapter_DbTable` adapter. The adapter works by allowing us to define specific fields on that table used for the login process, i.e. the username and password fields. It also lets us define an operation needed for these fields which in our case is primarily the need to apply SHA256 to any password before comparing it to the database stored value. Unfortunately since MySQL and SHA256 aren't happy campers I've explicitly applied the hashing here.

Once the Adapter is configured we need only add two steps. Firstly we tell the Adapter to attempt authentication using the values provided from our login form, and secondly we store the authenticated author's data for future reference elsewhere in the application.

Since the only authenticated user is the author, on a successful login we'll redirect them to the Administration Module's index page.

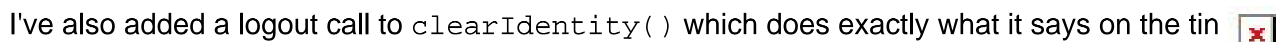
Here's the updated version of our AuthorController class.

```php
<?php
class AuthorController extends Zend_Controller_Action
{

    public function loginAction()
    {
        $form = new ZFBlog_Form_AuthorLogin;
        if (!$this->getRequest()->isPost() || !$form->isValid($_POST)) {
            $this->view->loginForm = $form;
            return;
        }
        $values = $form->getValues();
        // Setup DbTable adapter
        $adapter = new Zend_Auth_Adapter_DbTable(
            Zend_Db_Table::getDefaultAdapter() // set earlier in Bootstrap
        );
        $adapter->setTableName('authors');
        $adapter->setIdentityColumn('username');
        $adapter->setCredentialColumn('password');
        $adapter->setIdentity($values['name']);
        $adapter->setCredential(
            hash('SHA256', $values['password'])
        );
        // authentication attempt
        $auth = Zend_Auth::getInstance();
        $result = $auth->authenticate($adapter);
        // authentication succeeded
        if ($result->isValid()) {
            $auth->getStorage()
                ->write($adapter->getResultRowObject(null, 'password'));
            $this->_helper->redirector('index', 'index', 'admin');
        } else { // or not! Back to the login page!
            $this->view->failedAuthentication = true;
            $this->view->loginForm = $form;
        }
    }
    public function logoutAction()
    {
        Zend_Auth::getInstance()->clearIdentity();
        $this->_helper->redirector('index', 'index');
    }

}
```

As you can see, using Zend_Auth is pretty easy. Authentication is driven by the Adapter in use, and performed by a quick call to the singleton (there can only be one such object globally) Zend_Auth instance. The rest is storing the credentials and setting View result values for our template to make use of.

Note that the `getResultRowObject()` method parameters tell the storage driver not to remember the password hash. Personally I don't want the hash leaving the database if I can help it - it serves no purpose

and is the kind of user data you might consider not persisting outside authentication. You could drop that entire line if you didn't care if passwords are stored or not since Zend_Auth will store the identity by default.

I've also added a logout call to `clearIdentity()` which does exactly what it says on the tin  . Might be useful when you're testing at home - I'll add a logout link to the interface another time.

A little feedback would probably be appreciated by the author if their login attempt failed, so let's amend our login View at `/application/views/scripts/author/login.phtml` accordingly.

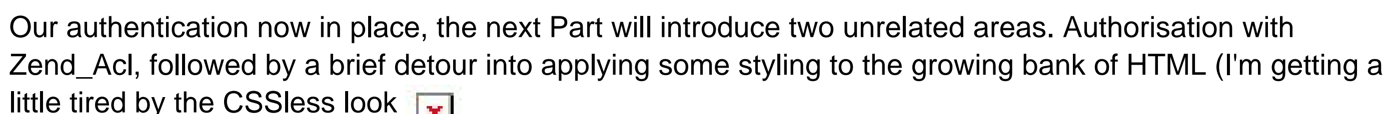
```
<h2>Authentication</h2>
<p>Enter your author name and password below.</p>
<?php if (isset($this->failedAuthentication)): ?>
   <p class="error">Sorry, but the credentials supplied could not be authenticated. Please try again.</p>
<?php endif; ?>
<?php echo $this->loginForm->render() ?>
```

Go ahead, open a browser and take the new login form for a test run.

Conclusion

Since authentication is really simple with the Zend_Auth component, the bulk of this installment concerned using Zend_Form. I really do advise taking a form apart before calling on Zend_Form because you really need to know how a form's structure can be built up in layers using the Zend_Form decorators.

If anything, the decorators are the main obstacle you'll meet using Zend_Form since almost everything else is straight forward once you put in place a bit of structure. At all costs avoid the umpteen line approach to Zend_Form usage where elements, decorators and validators are tied so closely together maintenance is a lost cause. One point I didn't cover was adding custom Element classes - in a form heavy application that could save you even more typing.

Our authentication now in place, the next Part will introduce two unrelated areas. Authorisation with Zend_Acl, followed by a brief detour into applying some styling to the growing bank of HTML (I'm getting a little tired by the CSSless look ).

Note: The source code for this entry is available to browse, or checkout with subversion, from http://svn.astrumfutura.org/zfblog/tags/Part6. The full source code for the entire application (as it exists thus far) from http://svn.astrumfutura.org/zfblog. The source code now contains some additional directories utilised to build distributable versions of the source code using Phing.

Posted by PÃ¡draic Brady in PHP General, PHP Security, Zend Framework at 06:55

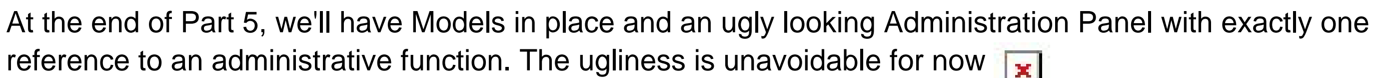
Thursday, May 1. 2008

An Example Zend Framework Blog Application - Part 5: Creating Models with Zend_Db and adding an Administration Module

Today's entry adds a few more layers to our slowly growing blog application. First of all I decided to add an Entries Model and Authors Model to the mix, primarily to get ready for when we can add new entries to our blog. This leads to where we can create new Entries; we add an Administration Module to the application with it's own distinct Layout.

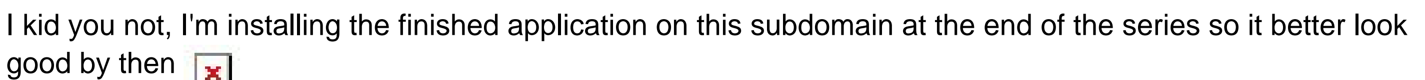
Previously: [Part 4: Setting the Design Stage with Blueprint CSS Framework and Zend_Layout]

At the end of Part 5, we'll have Models in place and an ugly looking Administration Panel with exactly one reference to an administrative function. The ugliness is unavoidable for now



. We'll have to start working on `/public/css/style.css` to make some custom changes to the default Blueprint CSS styling soon.

I kid you not, I'm installing the finished application on this subdomain at the end of the series so it better look good by then



.

Step 1: Creating a Database Schema

The schema for our blog's database will be aimed at MySQL. We're only starting with two tables to hold entries and authors, which may appear a little suspicious. The suspicion of course is due to the lack of two elements: Authentication and Authorisation. We'll cover both in Part 6 in the near future.

Here's the tables I came up with for now - ensure you have a local database set up, perhaps using phpMyAdmin, so you can just throw in this SQL to create the tables.

```
CREATE TABLE `entries` (
`id` int(11) NOT NULL auto_increment,
`title` varchar(200) collate utf8_unicode_ci NOT NULL,
`date` timestamp NOT NULL default '0000-00-00 00:00:00',
`author` varchar(60) collate utf8_unicode_ci NOT NULL default 'anonymous',
`author_id` int(11) NOT NULL default '0',
`body` text collate utf8_unicode_ci NOT NULL,
`extended_body` text collate utf8_unicode_ci NOT NULL,
`comment_count` int(4) NOT NULL default '0',
`last_modified` timestamp NOT NULL default CURRENT_TIMESTAMP on update
CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
FULLTEXT KEY `title` (`title`),
FULLTEXT KEY `body` (`body`),
FULLTEXT KEY `extended_body` (`extended_body`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

CREATE TABLE `authors` (
`id` int(11) NOT NULL auto_increment,
`realname` varchar(255) collate utf8_unicode_ci NOT NULL,
```

```
`username` varchar(20) collate utf8_unicode_ci NOT NULL,
`password` varchar(64) collate utf8_unicode_ci NOT NULL,
`email` varchar(128) collate utf8_unicode_ci NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

We'll likely only use the bare minimum of fields to start with. There are others such as `comment_count` in `entries` we can't use yet since we don't have a commenting system in place, and likely more just not included above yet. Obviously the input page for entries can be anything from a simple set of form fields, to a three page monstrosity!

If you're curious about why we're using MyISAM for the entries table instead of InnoDB as for authors, the simple reason is that full-text indices are only available on MyISAM tables to enable search functionality using the `MATCH()` function in MySQL. We've added full-text indices for the title, body, and extended_body fields.

Step 2: Adding the Models

To interact with our newly created database tables, we're going to need some Models. Back in our previous excursion into explaining Model-View-Controller (MVC) we described that a Model is basically a representation of application state - i.e. data persisted between requests often in a database. The Model for a database in the Zend_Framework is usually captured as a Class inheriting from `Zend_Db_Table`.

The Model for our `entries` table will be maintained in an `Entries` class in `/application/models/Entries.php`:

```php
<?php
class Entries extends Zend_Db_Table
{
    protected $_name = 'entries';
}
```

Similarly we will have `/application/models/Authors.php`:

```php
<?php
class Authors extends Zend_Db_Table
{
    protected $_name = 'authors';
}
```

It really is that simple for now. `Zend_Db_Table` even assumes the primary key is "id" unless otherwise defined. If you really want to know what the fuss about Models is, it's all in the methods you add to this class. For now, it's a blank slate referencing the database's table name. But you can imagine a few business logic rules you could consider adding here (hint: processing of entry bodies before saving using HTMLPurifier). The main idea to keep in mind is that if you're doing something to data from a Model, and it's potentially useable in more than one Controller, then chances are you're better off adding it as a Model method. Why make your Controllers fat and bloated? The more bloat they contain, the harder it becomes to see the application's workflow by reading them!

Step 3: Adding Database Credentials and a Default Database Adapter for Zend_Db

We have our database schema in place along with Models to represent it. Time to do something so our Model can actually interact with the database then

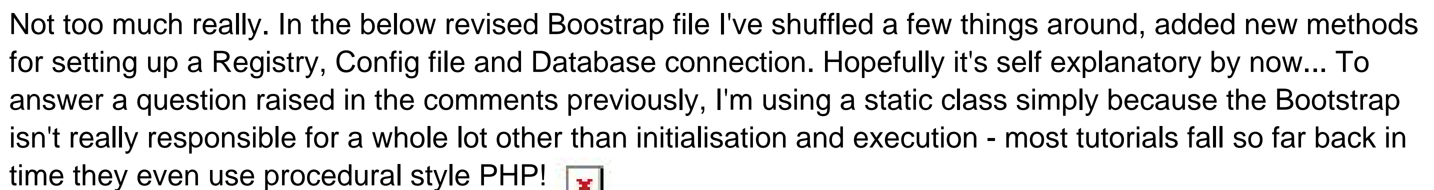
The first step is storing our database credentials in an editable file. Create a new file called `config.ini` in `/config` containing something along the lines of the following (edit for your personal credentials and database name):

```
[general]

;Database connection settings
db.adapter=PDO_MYSQL
db.host=localhost
db.username=root
db.password=passwd
db.dbname=zfblog
```

Note: The Subversion repository contains a template of the above file called `config.ini.example`. This way I can change my own `config.ini` file (which is on the subversion ignore list for its parent directory) without committing its changes to the repository all the time! You will need to manually create a copy if running from Subversion.

By now you probably know the drill, and you're wondering just how far I mangled `/application/Bootstrap.php` to integrate database setup...

Not too much really. In the below revised Boostrap file I've shuffled a few things around, added new methods for setting up a Registry, Config file and Database connection. Hopefully it's self explanatory by now... To answer a question raised in the comments previously, I'm using a static class simply because the Bootstrap isn't really responsible for a whole lot other than initialisation and execution - most tutorials fall so far back in time they even use procedural style PHP!

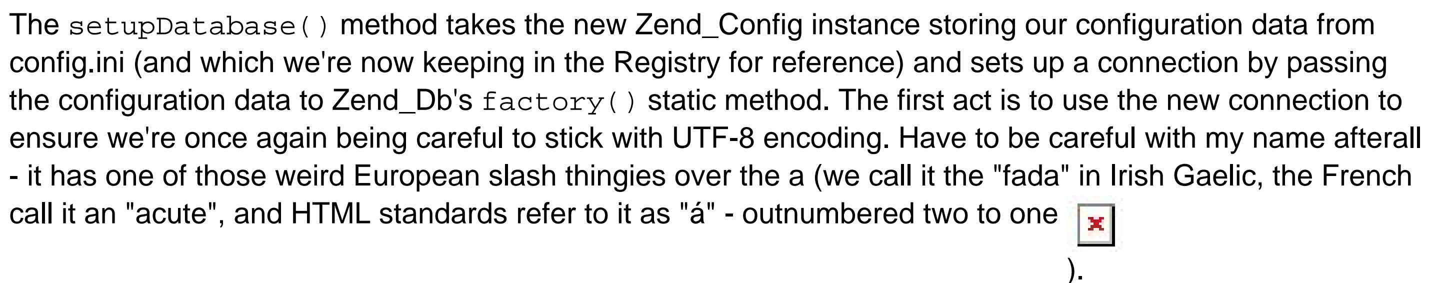


Static methods are also attractive because I do rely on not needing a discrete instance for other reasons - which we're not covering here since I'm not obsessing about Behaviour-Driven Development or TDD in this series.

```php
<?php
require_once 'Zend/Loader.php';
class Bootstrap
{
  public static $frontController = null;
  public static $root = '';
  public static $registry = null;

  public static function run()
  {
    self::prepare();
    $response = self::$frontController->dispatch();
    self::sendResponse($response);
  }

  public static function setupEnvironment()
  {
    error_reporting(E_ALL|E_STRICT);
```

```
    ini_set('display_errors', true);
    date_default_timezone_set('Europe/London');
    self::$root = dirname(dirname(<u>_FILE_</u>));
}

public static function prepare()
{
    self::setupEnvironment();
    Zend_Loader::registerAutoload();
    self::setupRegistry();
    self::setupConfiguration();
    self::setupFrontController();
    self::setupView();
    self::setupDatabase();
}

public static function setupFrontController()
{
    self::$frontController = Zend_Controller_Front::getInstance();
    self::$frontController->throwExceptions(true);
    self::$frontController->returnResponse(true);
    self::$frontController->setControllerDirectory(
        self::$root . '/application/controllers'
    );
    self::$frontController->setParam('registry', self::$registry);
}

public static function setupView()
{
    $view = new Zend_View;
    $view->setEncoding('UTF-8');
    $viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
    Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
    Zend_Layout::startMvc(
        array(
            'layoutPath' => self::$root . '/application/views/layouts',
            'layout' => 'common'
        )
    );
}

public static function sendResponse(Zend_Controller_Response_Http $response)
{
    $response->setHeader('Content-Type', 'text/html; charset=UTF-8', true);
    $response->sendResponse();
}
public static function setupRegistry()
{
    self::$registry = new Zend_Registry(array(), ArrayObject::ARRAY_AS_PROPS);
    Zend_Registry::setInstance(self::$registry);
}
public static function setupConfiguration()
{
```

```
        $config = new Zend_Config_Ini(
            self::$root . '/config/config.ini',
            'general'
        );
        self::$registry->configuration = $config;
    }
    public static function setupDatabase()
    {
        $config = self::$registry->configuration;
        $db = Zend_Db::factory($config->db->adapter, $config->db->toArray());
        $db->query("SET NAMES 'utf8'");
        self::$registry->database = $db;
        Zend_Db_Table::setDefaultAdapter($db);
    }

}
```

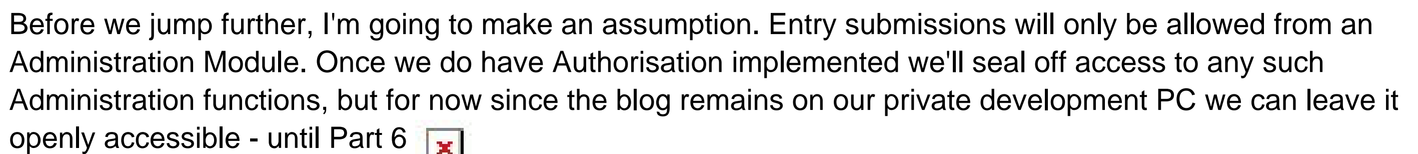
The `setupDatabase()` method takes the new Zend_Config instance storing our configuration data from config.ini (and which we're now keeping in the Registry for reference) and sets up a connection by passing the configuration data to Zend_Db's `factory()` static method. The first act is to use the new connection to ensure we're once again being careful to stick with UTF-8 encoding. Have to be careful with my name afterall - it has one of those weird European slash thingies over the a (we call it the "fada" in Irish Gaelic, the French call it an "acute", and HTML standards refer to it as "á" - outnumbered two to one ).

The connection object is stored to the Registry in case it's required later. It's usually needed as a last resort if we need it for something a Model isn't well suited for. The Registry is passed to the Front Controller in `setupFrontController()` as a user parameter. Finally, but not least, we make this new connection the default for Zend_Db_Table - available to all our Models.

It does look like a complicated web of steps, but honestly the code is pretty small for all this.

Step 4: Adding an Administration Module

We have the database, the Model, and the database connection.

Before we jump further, I'm going to make an assumption. Entry submissions will only be allowed from an Administration Module. Once we do have Authorisation implemented we'll seal off access to any such Administration functions, but for now since the blog remains on our private development PC we can leave it openly accessible - until Part 6  .

The Zend Framework allows for all Controllers and Views to be grouped into Modules relatively easily. Up to now we've been putting everything into their default locations without any thought of Modules, so it's high time we added one for Administration.

The first step is to introduce a new directory to our current directory structure called `admin` inside `/application`. You can also delete the previously suggested `modules` directory - we'll keep the directory tree a little flatter than I usually go with, although I can change this if readers prefer. Inside `admin`, add both a `controllers` and `views` directory. The `views` directory should in turn have `filters`, `helpers`, `layouts`

and `scripts` directories.

To allow our application divert requests to the new `admin` module, we also need to register its `controllers` directory with the Front Controller. This is a quick edit to our Bootstrap file at `/application/Boostrap.php` in the `setupFrontController()` method:

```
<?php
require_once 'Zend/Loader.php';
class Bootstrap
{
   public static $frontController = null;
   public static $root = '';
   public static $registry = null;

   public static function run()
   {
      self::prepare();
      $response = self::$frontController->dispatch();
      self::sendResponse($response);
   }

   public static function setupEnvironment()
   {
      error_reporting(E_ALL|E_STRICT);
      ini_set('display_errors', true);
      date_default_timezone_set('Europe/London');
      self::$root = dirname(dirname(<u>_FILE_</u>));
   }

   public static function prepare()
   {
      self::setupEnvironment();
      Zend_Loader::registerAutoload();
      self::setupRegistry();
      self::setupConfiguration();
      self::setupFrontController();
      self::setupView();
      self::setupDatabase();
   }

   public static function setupFrontController()
   {
      self::$frontController = Zend_Controller_Front::getInstance();
      self::$frontController->throwExceptions(true);
      self::$frontController->returnResponse(true);
      self::$frontController->setControllerDirectory(
         array(
            'default' => self::$root . '/application/controllers',
            'admin' => self::$root . '/application/admin/controllers'
         )
      );
      self::$frontController->setParam('registry', self::$registry);
   }
```

```
    public static function setupView()
    {
        $view = new Zend_View;
        $view->setEncoding('UTF-8');
        $viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
        Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
        Zend_Layout::startMvc(
            array(
                'layoutPath' => self::$root . '/application/views/layouts',
                'layout' => 'common'
            )
        );
    }

    public static function sendResponse(Zend_Controller_Response_Http $response)
    {
        $response->setHeader('Content-Type', 'text/html; charset=UTF-8', true);
        $response->sendResponse();
    }
    public static function setupRegistry()
    {
        self::$registry = new Zend_Registry(array(), ArrayObject::ARRAY_AS_PROPS);
        Zend_Registry::setInstance(self::$registry);
    }
    public static function setupConfiguration()
    {
        $config = new Zend_Config_Ini(
            self::$root . '/config/config.ini',
            'general'
        );
        self::$registry->configuration = $config;
    }
    public static function setupDatabase()
    {
        $config = self::$registry->configuration;
        $db = Zend_Db::factory($config->db->adapter, $config->db->toArray());
        $db->query("SET NAMES 'utf8'");
        self::$registry->database = $db;
        Zend_Db_Table::setDefaultAdapter($db);
    }

}
```

Nothing huge has changed. The only odd part perhaps, is that our previous controller directory is now assigned as a "default". This is a special Module key and you should never omit it when setting up Modules since it is the ultimate fallback position for failed requests (after that it's to the ErrorController we'll add in the next Part of this tutorial series).

Our Administration module is going to have two controllers. An Index Controller which will default to displaying a list of available actions (e.g. create new blog entry), and an Entry Controller for adding new Entries (and later editing and deleting them). We'll create a new Index Controller at

/application/admin/controllers/IndexController.php containing the following:

```
<?php
class Admin_IndexController extends Zend_Controller_Action
{
   public function indexAction()
   {
   }
}
```

You'll notice that all Controllers not part of the default Module must have their class name prefixed with the Module name followed by an underscore. This merely ensures we have no annoying name conflicts between Modules over common Controller names. The indexAction method is completely empty - on purpose. The controller has nothing to do here except look pretty for a few microseconds until control passes to the ViewRenderer Action Helper to have the correct View rendered.

The most obvious next step, therefore, is adding a template for our new module's index page. Create index.phtml at /application/admin/views/scripts/index/index.phtml and we'll just add, for now, a simple listing of available functions. All one of them.

The first one we need of course, is a URL for adding a new Blog entry. We'll assume we're going to use an Entry Controller within the Admin Module:

```
<h2>Administration</h2>
<ul>
<li><a href="/admin/entry/add">New Entry</a></li>
</ul>
```

Note: Please do not forget the leading forward slash in relative URLs! That little character ensures all relative URLs remain relative to the base URL and not just get appended to the current URL (which is all prettied up).

Try opening up our budding Administration Panel using the URL http://zfblog/admin.

There's one small niggle here, which demonstrates another interesting facet of using Zend_Layout. The index page for Administration has that text filled right column. We don't need that - it's destined to hold Blog related stuff for public consumption. Let's make our Admin module utilise a slightly different layout! We'll add a narrower left column this time for future Administration functions.

Step 5: Adding an Administration Module specific Layout

So, add a new template admin.phtml to /application/admin/views/layouts/admin.phtml. This will be an exact duplicate of /application/views/layouts/common.phtml with a few column changes.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
   <meta name="language" content="en" />
```

```
    <title><?php echo $this->escape($this->title) ?></title>
    <link rel="stylesheet" href="/css/blueprint/screen.css" type="text/css" media="screen, projection"
>
    <link rel="stylesheet" href="/css/style.css" type="text/css" media="screen, projection">
    <link rel="stylesheet" href="/css/blueprint/print.css" type="text/css" media="print">
    <!--[if IE]><link rel="stylesheet" href="/css/blueprint/ie.css" type="text/css" media="screen,
projection"><![endif]-->
</head>
<body>

    <div class="container">
        <div class="block">
            <div id="zfHeader" class="column span-24">
                <h1>Lorem Ipsum</h1>
            </div>
        </div>
        <div class="block">
            <div id="zfExtraLeft" class="column span-5">
                <!-- We'll add the style to style.css later -->
                <div class="zfMenuLeft" style="margin-top: 3em;">
                    Lorem ipsum dolor sit amet
                    Lorem ipsum dolor sit amet
                    Lorem ipsum dolor sit amet
                    Lorem ipsum dolor sit amet
                </div>
            </div>

            <div id="zfContent" class="column span-18">

                <?php echo $this->layout()->content ?>
            </div>
        </div>
        <div class="block">
            <div id="zfFooter" class="span-24">
                <p>Copyright &copy; 2008 PÃ¡draic Brady</p>
            </div>
        </div>
    </div>
</body>
</html>
```

In order to get the Admin Module using this layout, we need to intercept the Zend_Layout object just before an application request is dispatched to any Controller. We can then alter the default configuration we have used in our Bootstrap file. Welcome to the world of the Front Controller Plugin!

A front controller plugin already exists called `Zend_Layout_Controller_Plugin_Layout` which contains a postDispatch() method which is where the Layout gets finally rendered. To switch Layouts, we can simply create a new class extending the existing plugin, and add a preDispatch() method to detect when the Admin Module has been requested and replace the Zend_Layout layout name and the path to the Module's `layouts` directory.

Front Controllers are very useful in this fashion for performing actions which have a dependency on the

Module, Controller or Actions being requested. In Part 6 of this series, we'll use another Front Controller Plugin to implement an Access Control List system for Authorisation using Zend_Acl which uses Module/Controller/Action names to check if the current user is authorised to access them.

To start, create a new directory tree within `/library` called `ZFBlog`. It will use the standard PEAR Convention directory structure for a class called `ZFBlog_Layout_Controller_Plugin_Layout`:

```
/library
    /ZFBlog
        /Layout
            /Controller
                /Plugin
                    /Layout.php
```

I suggest keeping an eye on ZFBlog. As you start to develop Zend Framework applications you will likely end up with two distinct types of additional classes. Zend Framework specific extensions and subclasses, and application specific classes. A lot of the time, such quirky additions can be reused in other applications. I've used this Layout plugin more than once

.

The contents of the Layout.php file could be as simple as:

```
<?php
class ZFBlog_Layout_Controller_Plugin_Layout extends Zend_Layout_Controller_Plugin_Layout
{

  public function preDispatch(Zend_Controller_Request_Abstract $request)
  {
    switch ($request->getModuleName())
    {
      case 'admin':
        $this->_moduleChange('admin');
    }
  }
  protected function _moduleChange($moduleName)
  {
    $this->getLayout()->setLayoutPath(
      dirname(dirname(
        $this->getLayout()->getLayoutPath()
      ))
      . DIRECTORY_SEPARATOR . $moduleName . '/views/layouts'
    );
    $this->getLayout()->setLayout($moduleName);
  }

}
```

The above plugin is really simple. Before a request is dispatched we get the Module name from the Request object and check it against our switch statement. If a match is found, we can reset the Zend_Layout layout name and path for that request, or if nothing matches we leave things as they stand.

If you are particularly astute at refactoring, you may feel explicitly mentioning "admin" in the class is a bad move - I'll leave it to readers to search for more flexible solutions handling multiple modules with different layouts.

Now, we have a new layout template and we have a plugin to utilise it when the Admin module is requested. Let's make sure this new class replaces the `Zend_Layout_Controller_Plugin_Layout` class referenced by default in Zend_Layout. Guess where that gets done?

The Bootstrap! Check out the revised setupView() method where we change the default plugin class. Note that since this project is utilising Autoloading, and we're storing the new plugin class in `/library` while also applying the PEAR Naming Convention, we require no other class inclusions and such. It just gets autoloaded when needed.

Note: The Zend Frameworks default autoloading function is only useful for classes and libraries strictly following the PEAR Conventions. I suggest moving non-PEAR Convention libraries and classes to a separate `vendor` directory parallel to `library`.

```
<?php
require_once 'Zend/Loader.php';
class Bootstrap
{
   public static $frontController = null;
   public static $root = '';
   public static $registry = null;

   public static function run()
   {
      self::prepare();
      $response = self::$frontController->dispatch();
      self::sendResponse($response);
   }

   public static function setupEnvironment()
   {
      error_reporting(E_ALL|E_STRICT);
      ini_set('display_errors', true);
      date_default_timezone_set('Europe/London');
      self::$root = dirname(dirname(<u>_FILE_</u>));
   }

   public static function prepare()
   {
      self::setupEnvironment();
      Zend_Loader::registerAutoload();
      self::setupRegistry();
      self::setupConfiguration();
      self::setupFrontController();
```

```
    self::setupView();
    self::setupDatabase();
}

public static function setupFrontController()
{
    self::$frontController = Zend_Controller_Front::getInstance();
    self::$frontController->throwExceptions(true);
    self::$frontController->returnResponse(true);
    self::$frontController->setControllerDirectory(
        array(
            'default' => self::$root . '/application/controllers',
            'admin' => self::$root . '/application/admin/controllers'
        )
    );
    self::$frontController->setParam('registry', self::$registry);
}

public static function setupView()
{
    $view = new Zend_View;
    $view->setEncoding('UTF-8');
    $viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
    Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
    Zend_Layout::startMvc(
        array(
            'layoutPath' => self::$root . '/application/views/layouts',
            'layout' => 'common',
            'pluginClass' => 'ZFBlog_Layout_Controller_Plugin_Layout'
        )
    );
}

public static function sendResponse(Zend_Controller_Response_Http $response)
{
    $response->setHeader('Content-Type', 'text/html; charset=UTF-8', true);
    $response->sendResponse();
}
public static function setupRegistry()
{
    self::$registry = new Zend_Registry(array(), ArrayObject::ARRAY_AS_PROPS);
    Zend_Registry::setInstance(self::$registry);
}
public static function setupConfiguration()
{
    $config = new Zend_Config_Ini(
        self::$root . '/config/config.ini',
        'general'
    );
    self::$registry->configuration = $config;
}
public static function setupDatabase()
{
```

```
        $config = self::$registry->configuration;
        $db = Zend_Db::factory($config->db->adapter, $config->db->toArray());
        $db->query("SET NAMES 'utf8'");
        self::$registry->database = $db;
        Zend_Db_Table::setDefaultAdapter($db);
    }

}
```

Go ahead and reload `http://zfblog/admin` in a browser. New module, new layout!

Conclusion

In this fifth installment in our series we've visited Models and Modules. By now I'd say people are probably wondering where all the good stuff is . Patience, Padawan. There is a lot to take in so far, and it only gets worse!

In Part 6, I'll be adding Authentication and Authorisation using Zend_Auth and Zend_Acl. By this point we'll have an Author, and we can spend Part 7 looking into how to save and retrieve blog entries using our Models, Zend_Form, and our pre-prepared Administration Module.

Questions are welcome in the comments - don't hold back!

Continue to: [Part 6: Introduction to Zend_Form and Authentication with Zend_Auth]

Note: The source code for this entry is available to browse, or checkout with subversion, from http://svn.astrumfutura.org/zfblog/tags/Part5. The full source code for the entire application (as it exists thus far) from http://svn.astrumfutura.org/zfblog.

Posted by PÃ¡draic Brady in PHP General, PHP Security, Zend Framework at 15:45